deckhand Documentation

Release 0.1

Deckhand Authors

Jul 07, 2020

Contents

Deckhand is a document-based configuration storage service built with auditability and validation in mind. It serves as the back-end storage service for Airship.

Deckhand's primary responsibilities include validating and storing YAML documents that are layered together to produce finalized documents, containing site configuration data, including sensitive data. Secrets can be stored using specialized secret storage management services like Barbican and later substituted into finalized or "rendered" documents.

The service understands a variety of document types, the combination of which describe the manner in which Deckhand renders finalized documents for consumption by other Airship services.

CHAPTER 1

Overview

1.1 Overview

Deckhand provides document revision management, storage and mutation functionality upon which the rest of the Airship components rely for orchestration of infrastructure provisioning. Deckhand understands declarative YAML documents that define, end-to-end, the configuration of sites: from the hardware – encompassing network topology and hardware and host profile information – up to the software level that comprises the overcloud.

1.1.1 Core Responsibilities

- *revision history* - improves auditability and enables services to provide functional validation of a well-defined collection of documents that are meant to operate together
- *validation* - allows services to implement and register different kinds of validations and report errors
- *buckets* - allow documents to be owned by different services, providing write protections around collections of documents
- *layering* - helps reduce duplication in configuration while maintaining auditability across many sites
- *substitution* - provides separation between secret data and other configuration data, while also providing a mechanism for documents to share data among themselves

1.1.2 Revision History

Like other version control software, Deckhand allows users to track incremental changes to documents via a revision history, built up through individual payloads to Deckhand, each forming a separate revision. Each revision, in other words, contains its own set of immutable documents: Creating a new revision maintains the existing revision history.

For more information, see the *Revision History* section.

1.1.3 Validation

For each created revision, built-in *Document Types* are automatically validated. Validations are always stored in the database, including detailed error messages explaining why validation failed, to help deployers rectify syntactical or semantical issues with configuration documents. Regardless of validation failure, a new revision is **always** created, except when the documents are completely malformed.

Deckhand validation functionality is extensible via `DataSchema` documents, allowing the `data` sections of registered document types to be subjected to user-provided JSON schemas.

Note: While Deckhand ingests YAML documents, internally it translates them to Python objects and can use JSON schemas to validate those objects.

For more information, see the *Document Validation* section.

1.1.4 Buckets

Collections of documents, called buckets, are managed together. All documents belong to a bucket and all documents that are part of a bucket must be fully specified together.

To create or update a new document in, e.g. bucket `mop`, one must PUT the entire set of documents already in `mop` along with the new or modified document. Any documents not included in that PUT will be automatically deleted in the created revision.

Each bucket provides write protections around a group of documents. That is, only the bucket that owns a collection of documents can manage those documents. However, documents can be read across different buckets and used together to render finalized configuration documents, to be consumed by other services like Armada, Drydock, Promenade or Shipyard.

In other words:

- Documents can be **read** from any bucket.

  This is useful so that documents from different buckets can be used together for layering and substitution.

- Documents can only be **written** to by the bucket that owns them.

  This is useful because it offers the concept of ownership to a document in which only the bucket that owns the document can manage it.

Todo: Deckhand should offer RBAC (Role-Based Access Control) around buckets. This will allow deployers to control permissions around who can write or read documents to or from buckets.

Note: The best analogy for a bucket is a folder. Like a folder, which houses files and offers read and write permissions, a bucket houses documents and offers read and write permissions around them.

A bucket is **not** akin to a repository, because a repository has its own distinct revision history. A bucket, on the other hand, shares its revision history with every other bucket.

1.1.5 Layering

Layering provides a restricted data inheritance model intended to help reduce duplication in configuration. A `LayeringPolicy` can be created to declare the order of inheritance via layers for documents. Parent documents

can provide common data to child documents, who can override their parent data or tweak it in order to achieve more nuanced configuration that builds on top of common configurations.

For more information, see the *Document Layering* section.

1.1.6 Substitution

Substitution is a mechanism for documents to share data among themselves. It is particularly useful for documents that possess secrets to be stored securely and on demand provide the secrets to documents that need them. However, substitution can also apply to any data, not just secrets.

For more information, see the *Document Substitution* section.

1.1.7 Replacement

Document replacement provides an advanced mechanism for reducing the overhead with data duplication across multiple documents.

For more information, see the *Document Replacement* section.

CHAPTER 2

User's Guide

2.1 User's Guide

2.1.1 Getting Started

Pre-requisites

- tox

  To install tox run:

```
$ [sudo] apt-get install tox
```

- PostgreSQL

  Deckhand only supports PostgreSQL. Install it by running:

```
$ [sudo] apt-get update
$ [sudo] apt-get install postgresql postgresql-contrib
```

Quickstart

SQLite

The guide below provides details on how to run Deckhand quickly using SQLite.

Docker can be used to quickly instantiate the Deckhand image. After installing Docker, create a basic configuration file:

```
$ tox -e genconfig
```

Resulting deckhand.conf.sample file is output to :path:etc/deckhand/deckhand.conf.sample

Move the sample configuration file into a desired directory (i.e. `$CONF_DIR`).

Set the database string in the configuration file to `sqlite://`

```
[database]

#
# From oslo.db
#

# The SQLAlchemy connection string to use to connect to the database.
# (string value)
connection = sqlite://
```

Finally, run Deckhand via Docker:

```
$ [sudo] docker run --rm \
    --net=host \
    -p 9000:9000 \
    -v $CONF_DIR:/etc/deckhand \
    quay.io/airshipit/deckhand:latest-ubuntu_bionic
```

PostgreSQL

The guide below provides details on how to run Deckhand quickly using PostgreSQL.

Docker can be used to quickly instantiate the Deckhand image. After installing Docker, create a basic configuration file:

```
$ tox -e genconfig
```

Resulting deckhand.conf.sample file is output to :path:etc/deckhand/deckhand.conf.sample

Move the sample configuration file into a desired directory (i.e. `$CONF_DIR`).

At a minimum the `[database].connection` config option must be set. Provide it with a PostgreSQL database connection. Or to conveniently create an ephemeral PostgreSQL DB run:

```
$ eval `pifpaf run postgresql`
```

Substitute the connection information (which can be retrieved by running `export | grep PIFPAF_POSTGRESQL_URL`) into the config file inside `etc/deckhand/deckhand.conf.sample`:

```
[database]

#
# From oslo.db
#

# The SQLAlchemy connection string to use to connect to the database.
# (string value)
connection = postgresql://localhost/postgres?host=/tmp/tmpsg6tn3l9&port=9824
```

Run an update to the Database to bring it to the current code level:

```
$ [sudo] docker run --rm \
    --net=host \
    -v $CONF_DIR:/etc/deckhand \
    quay.io/airshipit/deckhand:latest-ubuntu_bionic\
    alembic upgrade head
```

Finally, run Deckhand via Docker:

```
$ [sudo] docker run --rm \
    --net=host \
    -p 9000:9000 \
    -v $CONF_DIR:/etc/deckhand \
    quay.io/airshipit/deckhand:latest-ubuntu_bionic
```

To kill the ephemeral DB afterward:

```
$ pifpaf_stop
```

Manual Installation

Note: The commands below assume that they are being executed from the root Deckhand directory.

Install dependencies needed to spin up Deckhand via `uwsgi`:

```
$ [sudo] pip install uwsgi
$ virtualenv -p python3 /var/tmp/deckhand
$ . /var/tmp/deckhand/bin/activate
$ pip install -r requirements.txt -r test-requirements.txt
$ python setup.py install
```

Afterward, create a sample configuration file automatically:

```
$ tox -e genconfig
```

Resulting deckhand.conf.sample file is output to :path:etc/deckhand/deckhand.conf.sample

Create the directory `/etc/deckhand` and copy the config file there:

```
$ [sudo] cp etc/deckhand/deckhand.conf.sample /etc/deckhand/deckhand.conf
```

To specify an alternative directory for the config file, run:

```
$ export DECKHAND_CONFIG_DIR=<PATH>
$ [sudo] cp etc/deckhand/deckhand.conf.sample ${DECKHAND_CONFIG_DIR}/deckhand.conf
```

To conveniently create an ephemeral PostgreSQL DB run:

```
$ eval `pifpaf run postgresql`
```

Retrieve the environment variable which contains connection information:

```
$ export | grep PIFPAF_POSTGRESQL_URL
declare -x PIFPAF_POSTGRESQL_URL="postgresql://localhost/postgres?host=/tmp/
↪tmpsg6tn3l9&port=9824"
```

Substitute the connection information into the config file in `${DECKHAND_CONFIG_DIR}`:

```
[database]

#
# From oslo.db
#

# The SQLAlchemy connection string to use to connect to the database.
# (string value)
connection = postgresql://localhost/postgres?host=/tmp/tmpsg6tn3l9&port=9824
```

Finally, run Deckhand:

```
# Perform DB migrations
$ ./entrypoint.sh alembic upgrade head
# Instantiate the Deckhand server
$ ./entrypoint.sh server
```

To kill the ephemeral DB afterward:

```
$ pifpaf_stop
```

Development Mode

Development mode means running Deckhand without Keystone authentication. Note that enabling development mode will effectively disable all authN and authZ in Deckhand.

To enable development mode, add the following to the `deckhand.conf` inside `$CONF_DIR`:

```
[DEFAULT]
development_mode = True
```

After, from the command line, execute:

```
$ [sudo] docker run --rm \
    --net=host \
    -p 9000:9000 \
    -v $CONF_DIR:/etc/deckhand \
    quay.io/airshipit/deckhand:latest-ubuntu_bionic server
```

Development Utilities

Deckhand comes equipped with many utilities useful for developers, such as unit test or linting jobs.

Many of these commands require that `tox` be installed. To do so, run:

```
$ pip3 install tox
```

To run the Python linter, execute:

```
$ tox -e pep8
```

To run unit tests, execute:

```
$ tox -e py35
```

To run the test coverage job:

```
$ tox -e coverage
```

To run security checks via Bandit execute:

```
$ tox -e bandit
```

To build all Deckhand charts, execute:

```
$ make charts
```

To generate sample configuration and policy files needed for Deckhand deployment, execute (respectively):

```
$ tox -e genconfig
$ tox -e genpolicy
```

For additional commands, reference the `tox.ini` file for a list of all the jobs.

Database Model Updates

Deckhand utilizes Alembic to handle database setup and upgrades. Alembic provides a straightforward way to manage the migrations necessary from one database structure version to another through the use of scripts found in deckhand/alembic/versions.

Setting up a migration can be automatic or manual. The Alembic documentation provides instructions for how to create a new migration.

Creating automatic migrations requires that the Deckhand database model is updated in the source code first. With that database model in the code, and pointing to an existing Deckhand database structure, Alembic can produce the steps necessary to move from the current version to the next version.

One way of creating an automatic migration is to deploy a development Deckhand database using the pre-updated data model and following the following steps:

```
Navigate to the root Deckhand directory
$ export DH_ROOT=$(pwd)
$ mkdir ${DH_ROOT}/alembic_tmp

Create a deckhand.conf file that will have the correct DB connection string.
$ tox -e genconfig
$ cp ${DH_ROOT}/etc/deckhand/deckhand.conf.sample ${DH_ROOT}/alembic_tmp/deckhand.conf

Update the connection string to the deckhand db instance e.g.::

  [Database]
  connection = postgresql+psycopg2://deckhand:password@postgresql.airship.svc.cluster.
↪local:5432/deckhand

$ export DECKHAND_CONFIG_DIR=${DH_ROOT}/alembic_tmp
$ alembic revision --autogenerate -m "The short description for this change"

$ rm -r ${DH_ROOT}/alembic_tmp
```

This will create a new .py file in the deckhand/alembic/versions directory that can then be modified to indicate exact steps. The generated migration should always be inspected to ensure correctness.

Migrations exist in a linked list of files (the files in versions). Each file is updated by Alembic to reference its revision linkage. E.g.:

```
# revision identifiers, used by Alembic.
revision = '918bbfd28185'
down_revision = None
branch_labels = None
depends_on = None
```

Any manual changes to this linkage must be approached carefully or Alembic will fail to operate.

Troubleshooting

The error messages are included in bullets below and tips to resolution are included beneath each bullet.

- "FileNotFoundError: [Errno 2] No such file or directory: '/etc/deckhand/api-paste.ini'"

  Reason: this means that Deckhand is trying to instantiate the server but failing to do so because it can't find an essential configuration file.

  Solution:

  ```
  $ cp etc/deckhand/deckhand.conf.sample /etc/deckhand/deckhand.conf
  ```

  This copies the sample Deckhand configuration file to the appropriate directory.

- For any errors related to `tox`:

  Ensure that `tox` is installed:

  ```
  $ [sudo] apt-get install tox -y
  ```

- For any errors related to running `tox -e py35`:

  Ensure that `python3-dev` is installed:

  ```
  $ [sudo] apt-get install python3-dev -y
  ```

2.1.2 Revision History

Revision History

Documents will be ingested in batches which will be given a revision index. This provides a common language for describing complex validations on sets of documents.

Revisions can be thought of as commits in a linear git history, thus looking at a revision includes all content from previous revisions.

Revision Diffing

By maintaining a linear history of all the documents in each revision, Deckhand is able to diff different revisions together to report what has changed across revisions, allowing external services to determine whether the Deckhand configuration undergone any changes since the service last queried the Deckhand API.

The revision difference is calculated by comparing the *overall* difference across all the documents in the buckets associated with the two revisions that are diffed. For example, if a bucket shared between two revisions contains two documents, and between the first revision and the second revision, if only one of those two documents has been modified, the bucket itself is tagged as `modified`. For more information about revision diffing, please reference the *Deckhand API Documentation*.

Revision DeepDiffing

Revision DeepDiffing is an extended version of existing revision diff api. When any bucket state gets changed to `modified`, It shows deep difference between revisions. DeepDiffing resultset will consist of `document_added`, `document_deleted` and `document_changed` count and details. For more information about revision deepdiffing, please reference the *Deckhand API Documentation*.

Revision Rollback

As all the changes to documents are maintained via revisions, it is possible to rollback the latest revision in Deckhand to a prior revision. This behavior can be loosely compared to a `git rebase` in which it is possible to squash the latest revision in order to go back to the previous revision. This behavior is useful for undoing accidental changes and returning to a stable internal configuration.

2.1.3 Documents

All configuration data is stored entirely as structured documents, for which schemas must be registered. Documents satisfy the following use cases:

- layering - helps reduce duplication in configuration while maintaining auditability across many sites
- substitution - provides separation between secret data and other configuration data, while allowing a simple interface for clients
- revision history - improves auditability and enables services to provide functional validation of a well-defined collection of documents that are meant to operate together
- validation - allows services to implement and register different kinds of validations and report errors

Detailed documentation for *Document Layering*, *Document Substitution*, *Revision History* and *Document Validation* should be reviewed for a more thorough understanding of each concept.

Document Format

The document format is modeled loosely after Kubernetes practices. The top level of each document is a dictionary with 3 keys: `schema`, `metadata`, and `data`.

- `schema` - Defines the name of the JSON schema to be used for validation. Must have the form: `<namespace>/<kind>/<version>`, where the meaning of each component is:
  - `namespace` - Identifies the owner of this type of document. The values `deckhand` and `metadata` are reserved for internal use.
  - `kind` - Identifies a type of configuration resource in the namespace.
  - `version` - Describe the version of this resource, e.g. `v1`.
- `metadata` - Defines details that Deckhand will inspect and understand. There are multiple schemas for this section as discussed below. All the various types of metadata include a `metadata.name` field which must be unique for each document `schema`.

- `data` - Data to be validated by the schema described by the `schema` field. Deckhand only interacts with content here as instructed to do so by the `metadata` section. The form of this section is considered to be completely owned by the `namespace` in the `schema`.

At the **database** level, documents are uniquely identified by the combination of:

1. `metadata.name`
2. `schema`
3. `metadata.layeringDefinition.layer`

This means that raw revision documents – which are persisted in Deckhand's database – require that the combination of all 3 parameters be unique.

However, **post-rendered documents** are only uniquely identified by the combination of:

1. `metadata.name`
2. `schema`

Because collisions with respect to the third parameter – `metadata.layeringDefinition.layer` – can only occur with *Document Replacement*. But after document rendering, the replacement-parent documents are never returned.

Below is a fictitious example of a complete document, which illustrates all the valid fields in the `metadata` section:

```
---
schema: some-service/ResourceType/v1
metadata:
  schema: metadata/Document/v1
  name: unique-name-given-schema
  storagePolicy: cleartext
  labels:
    genesis: enabled
    master: enabled
  layeringDefinition:
    abstract: true
    layer: region
    parentSelector:
      required_key_a: required_label_a
      required_key_b: required_label_b
    actions:
      - method: merge
        path: .path.to.merge.into.parent
      - method: delete
        path: .path.to.delete
  substitutions:
    - dest:
        path: .substitution.target
      src:
        schema: another-service/SourceType/v1
        name: name-of-source-document
        path: .source.path
data:
  path:
    to:
      merge:
        into:
          parent:
            foo: bar
```

(continues on next page)

(continued from previous page)

```
        ignored:
          data: here
substitution:
  target: null
```

Document Metadata

There are 2 supported kinds of document metadata. Documents with `Document` metadata are the most common, and are used for normal configuration data. Documents with `Control` metadata are used to customize the behavior of Deckhand.

schema: metadata/Document/v1

This type of metadata allows the following metadata hierarchy:

- `name` - string, required - Unique within a revision for a given `schema` and `metadata.layeringDefinition.layer`.
- `storagePolicy` - string, required - Either `cleartext` or `encrypted`. If `encyrpted` is specified, then the `data` section of the document will be stored in a secure backend (likely via OpenStack Barbican). `metadata` and `schema` fields are always stored in cleartext. More information on document encryption is available *here*.
- `layeringDefinition` - dict, required - Specifies layering details. See the Layering section below for details.
  - `abstract` - boolean, required - An abstract document is not expected to pass schema validation after layering and substitution are applied. Non-abstract (concrete) documents are.
  - `layer` - string, required - References a layer in the `LayeringPolicy` control document.
  - `parentSelector` - labels, optional - Used to construct document chains for executing merges.
  - `actions` - list, optional - A sequence of actions to apply this documents data during the merge process. * `method` - string, required - How to layer this content. * `path` - string, required - What content in this document to layer onto parent content.
- `substitutions` - list, optional - A sequence of substitutions to apply. See the Substitutions section for additional details.
  - `dest` - dict, required - A description of the inserted content destination.
    - `path` - string, required - The JSON path where the data will be placed into the `data` section of this document.
    - `pattern` - string, optional - A regex to search for in the string specified at `path` in this document and replace with the source data
  - `src` - dict, required - A description of the inserted content source.
    - `schema` - string, required - The `schema` of the source document.
    - `name` - string, required - The `metadata.name` of the source document.
    - `path` - string, required - The JSON path from which to extract data in the source document relative to its `data` section.

schema: metadata/Control/v1

This schema is the same as the `Document` schema, except it omits the `storagePolicy`, `layeringDefinition`, and `substitutions` keys, as these actions are not supported on `Control` documents.

The complete list of valid `Control` document kinds is specified below along with descriptions of each document kind.

Document Abstraction

Document abstraction can be compared to an abstract class in programming languages: The idea is to declare an abstract base class used for declaring common data to be overridden and customized by subclasses. In fact, this is the predominant use case for document abstraction: Defining base abstract documents that other concrete (non-abstract) documents can layer with.

An abstract document is a document whose `metadata.abstract` property is True. A concrete document is a document whose `metadata.abstract` property is False. Concrete and non-abstract are terms that are used interchangeably.

In Deckhand, document abstraction has certain implications:

- An abstract document, like all other documents, will be persisted in Deckhand's database and will be subjected to *Revision History*.
- However, abstract documents are **not** returned by Deckhand's `rendered-documents` endpoint: That is, rendered documents never include abstract documents.
- Concrete documents **can** layer with abstract documents – and this is encouraged.
- Abstract documents **can** layer with other documents as well – but unless a concrete document layers with or substitutes from the resultant abstract document, no meaningful data will be returned via rendering, as only concrete documents are returned.
- Likewise, abstract documents **can** substitute from other documents. The same reasoning as the bullet point above applies.
- However, abstract documents **cannot** be used as substitution sources. Only concrete documents may be used as substitution sources.

2.1.4 Document Types

Application Documents

Application documents are those whose `metadata.schema` begins with `metadata/Document`. These documents define all the data that make up a site deployment, including but not limited to: networking, hardware, host, bare metal, software, etc. site information. Prior to ingestion by Deckhand, application documents are known as "raw documents". After rendering, they are known as "rendered documents". Application documents are subject to the following *Document Rendering* operations:

- *Data Encryption*
- *Document Layering*
- *Document Substitution*
- *Document Replacement*

Control Documents

Control documents (documents which have `metadata.schema` of `metadata/Control/v1`), are special, and are used to control the behavior of Deckhand at runtime. Control documents are immutable so any document mutation or manipulation does not apply to them.

Control documents only exist to control how *Application Documents* are validated and rendered.

Note: Unlike *Application Documents*, control documents do not require `storagePolicy` or `layeringDefinition` properties; in fact, it is recommended that such properties not be used for control documents. Again, this is because such documents should not themselves undergo layering, substitution or encryption. It is not meaningful to treat them like normal documents. See *Validation Schemas* for more information on required document properties.

Only the following types of control documents are allowed:

DataSchema

`DataSchema` documents are used by various services to register new schemas that Deckhand can use for validation. No `DataSchema` documents with names beginning with `deckhand/` or `metadata/` are allowed. The `metadata.name` field of each `DataSchema` document references the top-level `schema` of *Application Documents*: when there is a match between both values, the `data` section of all *Application Documents* is validated against the JSON schema found in the matching `DataSchema` document.

The JSON schema definition is found in the `data` key of each `DataSchema` document. The entire `data` section of the target document is validated.

The following is an example of a sample `DataSchema` document, whose `data` section features a simplistic JSON schema:

```
---
# This specifies the official JSON schema meta-schema.
schema: deckhand/DataSchema/v1
metadata:
  schema: metadata/Control/v1
  name: promenade/Node/v1  # Specifies the documents to be used for validation.
  labels:
    application: promenade
data:  # Valid JSON Schema is expected here.
  $schema: http://blah
  properties:
    foo:
      enum:
        - bar
        - baz
        - qux
  required:
    - foo
...
```

The JSON schema abvove requires that the `data` section of *Application Documents* that match this `DataSchema` have a property called `foo` whose value must be one of: "bar", "baz", or "qux".

Reference the JSON schema documentation for more information on writing correct schemas.

LayeringPolicy

This document defines the strict order in which documents are layered together from their component parts.

Only one `LayeringPolicy` document can exist within the system at any time. It is an error to attempt to insert a new `LayeringPolicy` document if it has a different `metadata.name` than the existing document. If the names match, it is treated as an update to the existing document.

Note: In order to create a new `LayeringPolicy` document in Deckhand, submit an **empty** payload via `PUT /buckets/{bucket_name}/documents`. Afterward, submit another request containing the new batch of documents, including the new `LayeringPolicy`.

This document defines the strict order in which documents are merged together from their component parts. An error is raised if a document refers to a layer not specified in the `LayeringPolicy`.

Below is an example of a `LayeringPolicy` document:

```
---
schema: deckhand/LayeringPolicy/v1
metadata:
  schema: metadata/Control/v1
  name: layering-policy
data:
  layerOrder:
    - global
    - site-type
    - region
    - site
    - force
...
```

In the `LayeringPolicy` above, a 5-tier `layerOrder` is created, in which the topmost layer is `global` and the bottommost layer is `force`. This means that `global` constitutes the "base" layer onto which other documents belonging to sub-layers can be layered. In practice, this means that documents with `site-type` can layer with documents with `global` and documents with `region` can layer with documents with `site-type`, etc.

Note that in the absence of any document belonging to an "intermediate" layer, base layers can layer with "interspersed" sub-layers, no matter the number of layers between them. This means that a document with layer `force` could layer with a document with layer `global`, *provided* no document exists with a layer of `site-type`, `region`, or `site`. For more information about document layering, reference the *Document Layering* documentation.

ValidationPolicy

Unlike `LayeringPolicy`, many `ValidationPolicy` documents are allowed. This allows services to check whether a particular revision (described below) of documents meets a configurable set of validations without having to know up front the complete list of validations.

Each validation `name` specified here is a reference to data that is POSTable by other services. Names beginning with `deckhand` are reserved for internal use. See the Validation section below for more details.

Since validations may indicate interactions with external and changing circumstances, an optional `expiresAfter` key may be specified for each validation as an ISO8601 duration. If no `expiresAfter` is specified, a successful validation does not expire. Note that expirations are specific to the combination of `ValidationPolicy` and validation, not to each validation by itself.

```
---
schema: deckhand/ValidationPolicy/v1
metadata:
  schema: metadata/Control/v1
  name: site-deploy-ready
data:
  validations:
    - name: deckhand-schema-validation
    - name: drydock-site-validation
      expiresAfter: P1W
    - name: promenade-site-validation
      expiresAfter: P1W
    - name: armada-deployability-validation
...
```

Provided Utility Document Kinds

These are documents that use the `Document` metadata schema, but live in the `deckhand` namespace.

Certificate

```
---
schema: deckhand/Certificate/v1
metadata:
  schema: metadata/Document/v1
  name: application-api
  storagePolicy: cleartext
data: |-
  -----BEGIN CERTIFICATE-----
  MIIDYDCCAkigAwIBAgIUKG41PW4VtiphzASAMY4/3hL8OtAwDQYJKoZIhvcNAQEL
  ...snip...
  P3WT9CfFARnsw2nKjnglQcwKkKLYip0WY2wh3FE7nrQZP6xKNaSRlh6p2pCGwwwH
  HkvVwA==
  -----END CERTIFICATE-----
...
```

CertificateAuthority

```
---
schema: deckhand/CertificateAuthority/v1
metadata:
  schema: metadata/Document/v1
  name: application-ca
  storagePolicy: cleartext
data: some-ca
...
```

CertificateAuthorityKey

```
---
schema: deckhand/CertificateAuthorityKey/v1
metadata:
  schema: metadata/Document/v1
  name: application-ca-key
  storagePolicy: encrypted
data: |-
  -----BEGIN CERTIFICATE-----
  MIIDYDCCAkigAwIBAgIUKG41PW4VtiphzASAMY4/3hL8OtAwDQYJKoZIhvcNAQEL
  ...snip...
  P3WT9CfFARnsw2nKjnglQcwKkKLYip0WY2wh3FE7nrQZP6xKNaSRlh6p2pCGwwwH
  HkvVwA==
  -----END CERTIFICATE-----
...
```

CertificateKey

```
---
schema: deckhand/CertificateKey/v1
metadata:
  schema: metadata/Document/v1
  name: application-api
  storagePolicy: encrypted
data: |-
  -----BEGIN RSA PRIVATE KEY-----
  MIIEpQIBAAKCAQEAx+m1+ao7uTVEs+I/Sie9YsXL0B9mOXFlzEdHX8P8x4nx78/T
  ...snip...
  Zf3ykIG8l71pIs4TGsPlnyeO6LzCWP5WRSh+BHnyXXjzx/uxMOpQ/6I=
  -----END RSA PRIVATE KEY-----
...
```

Passphrase

```
---
schema: deckhand/Passphrase/v1
metadata:
  schema: metadata/Document/v1
  name: application-admin-password
  storagePolicy: encrypted
data: some-password
...
```

PrivateKey

```
---
schema: deckhand/PrivateKey/v1
metadata:
  schema: metadata/Document/v1
```

(continues on next page)

(continued from previous page)

```
  name: application-private-key
  storagePolicy: encrypted
data: some-private-key
...
```

PublicKey

```
---
schema: deckhand/PublicKey/v1
metadata:
  schema: metadata/Document/v1
  name: application-public-key
  storagePolicy: cleartext
data: some-password
...
```

2.1.5 Data Encryption

Deckhand supports encrypting the `data` section of documents at-rest to secure sensitive data. This encryption behavior is triggered by setting `metadata.storagePolicy: encrypted`. It is solely the document author's responsibility to decide the appropriate storagePolicy for the data contained in the document.

Note: Note that encryption of document data incurs **runtime overhead** as the price of encryption is performance. As a general rule, the more documents with `storagePolicy: encrypted`, the longer it will take to render the documents, particularly because Barbican has a built-in restriction around retrieving only one encrypted payload a time. This means that if 50 documents have `storagePolicy: encrypted` within a revision, then Deckhand must perform 50 API calls to Barbican when rendering the documents for that revision.

Encrypted documents, like cleartext documents, are stored in Deckhand's database, except the `data` section of each encrypted document is replaced with a reference to Barbican.

Supported Data Types

Barbican supports encrypting any data type via its "opaque" secret type. Thus, Deckhand supports encryption of any data type by utilizing this secret type.

However, Deckhand will attempt to use Barbican's other secret types where possible. For example, Deckhand will use "public" for document types with kind `PublicKey`.

2.1.6 Data Redaction

Deckhand supports redacting sensitive document data, including:

- `data` section:
  - to avoid exposing the Barbican secret reference, in the case of the "GET documents" endpoint
  - to avoid exposing actual secret payloads, in the case of the "GET rendered-documents" endpoint
- `substitutions[n].src|dest` sections:

  - to avoid reverse-engineering where sensitive data is substituted from or into (in case the sensitive data is derived via *Document Substitution*)

Note: Document sections related to *Document Layering* do not require redaction because secret documents are *Control Documents*, which cannot be layered together.

See the *Deckhand API Documentation* for more information on how to redact sensitive data.

2.1.7 Document Validation

Validations

The validation system provides a unified approach to complex validations that require coordination of multiple documents and business logic that resides in consumer services.

Deckhand focuses on two types of validations: schema validations and policy validations.

Deckhand-Provided Validations

Deckhand provides a few internal validations which are made available immediately upon document ingestion. Deckhand's internal schema validations are defined as `DataSchema` documents.

Here is a list of internal validations:

- `deckhand-document-schema-validation` - All concrete documents in the revision successfully pass their JSON schema validations. Will cause this to report an error.

Externally Provided Validations

As mentioned, other services can report whether named validations that have been registered by those services as success or failure. `DataSchema` control documents are used to register a new validation mapping that other services can reference to verify whether a Deckhand bucket is in a valid configuration. For more information, refer to the `DataSchema` section in *Document Types*.

Validation Codes

- D001 - Indicates document sanity-check validation failure pre- or post-rendering. This means that the document structure is fundamentally broken.
- D002 - Indicates document post-rendering validation failure. This means that after a document has rendered, the document may fail validation. For example, if a `DataSchema` document for a given revision indicates that `.data.a` is a required field but a layering action during rendering deletes `.data.a`, then post-rendering validation will necessarily fail. This implies a conflict in the set of document requirements.

Schema Validations

Schema validations are controlled by two mechanisms:

1) Deckhand's internal schema validation for sanity-checking the formatting of the default documents that it understands. For example, Deckhand will check that a `LayeringPolicy`, `ValidationPolicy` or `DataSchema` adhere to the appropriate internal schemas.

2) Externally provided validations via `DataSchema` documents. These documents can be registered by external services and subject the target document's `data` section to *additional* validations, validations specified by the `data` section of the `DataSchema` document.

Validation Stages

Deckhand performs pre- and post-rendering validation on documents.

Pre-Rendering

Carried out during document ingestion.

For pre-rendering validation 3 types of behavior are possible:

1. Successfully validated documents are stored in Deckhand's database.
2. Failure to validate a document's basic properties will result in a 400 Bad Request error getting raised.
3. Failure to validate a document's schema-specific properties will result in a validation error created in the database, which can be later returned via the Validations API.

Post-Rendering

Carried out after rendering all documents.

For post-rendering validation, 2 types of behavior are possible:

1. Successfully validated post-rendered documents are returned to the user.
2. Failure to validate post-rendered documents results in a 500 Internal Server Error getting raised.

2.1.8 Validation Schemas

Below are the schemas Deckhand uses to validate documents.

Base Schema

- Base schema.

  Base JSON schema against which all Deckhand documents are validated.

Listing 1: Base schema that applies to all documents.

```
---
schema: deckhand/DataSchema/v1
metadata:
  name: deckhand/Base/v1
  schema: metadata/Control/v1
data:
  $schema: http://json-schema.org/schema#
  properties:
    schema:
      type: string
      pattern: ^[A-Za-z]+/[A-Za-z]+/v\d+$
```

(continues on next page)

(continued from previous page)

```
    metadata:
      # True validation of the metadata section will be done using
      # the schema specfied in the metadata section
      type: object
      properties:
        name:
          type: string
        schema:
          anyOf:
            - type: string
              pattern: ^metadata/Document/v\d+$
            - type: string
              pattern: ^metadata/Control/v\d+$
      additionalProperties: true
      required:
        - 'name'
        - 'schema'
    # This schema should allow anything in the data section
    data:
      type:
        - 'null'
        - 'string'
        - 'object'
        - 'array'
        - 'number'
        - 'boolean'
  additionalProperties: false
  required:
    - schema
    - metadata
    - data
```

This schema is used to sanity-check all documents that are passed to Deckhand. Failure to pass this schema results in a critical error.

Metadata Schemas

Metadata schemas validate the `metadata` section of every document ingested by Deckhand.

- `Metadata Control` schema.

  JSON schema against which the metadata section of each `metadata/Control` document type is validated. Applies to all static documents meant to configure Deckhand behavior, like LayeringPolicy, ValidationPolicy, and DataSchema documents.

  Listing 2: Schema for `metadata/Control` metadata document sections.

```
    labels:
      type: object
      additionalProperties:
        type: string
  additionalProperties: true
  required:
    - schema
    - name
```

(continues on next page)

(continued from previous page)

```
    # NOTE(felipemonteiro): layeringDefinition is not needed for any control
    # documents as neither LayeringPolicy, ValidationPolicy or DataSchema
    # documents are ever layered together.
```

- `Metadata Document` schema.

  JSON schema against which the metadata section of each `metadata/Document` document type is validated. Applies to all site definition documents or "regular" documents that require rendering.

  Listing 3: Schema for `metadata/Document` metadata document sections.

```
          - actions
    actions_requires_parent_selector:
      dependencies:
        # Requires that if actions are provided, then so too must
        # parentSelector.
        actions:
          required:
            - parentSelector
    substitution_dest:
      type: object
      properties:
        path:
          type: string
        pattern:
          type: string
        recurse:
          type: object
          properties:
            depth:
              type: integer
              minimum: -1
              # -1 indicates that the recursion depth is infinite. Refinements
              # to this value should be specified by the caller.
              default: -1
          required:
            - depth
      additionalProperties: false
      required:
        - path
  type: object
  properties:
    schema:
      type: string
      pattern: ^metadata/Document/v\d+$
    name:
      type: string
    labels:
      type: object
    replacement:
      type: boolean
    layeringDefinition:
      type: object
      properties:
        layer:
          type: string
```

(continues on next page)

(continued from previous page)

```
        abstract:
          type: boolean
        parentSelector:
          type: object
          minProperties: 1
        actions:
          type: array
          minItems: 1
          items:
            type: object
            properties:
              method:
                enum:
                  - replace
                  - delete
                  - merge
              path:
                type: string
            additionalProperties: false
            required:
              - method
              - path
    additionalProperties: false
    required:
      - 'layer'
    allOf:
      - $ref: "#/definitions/parent_selector_requires_actions"
      - $ref: "#/definitions/actions_requires_parent_selector"
  substitutions:
    type: array
    items:
      type: object
      properties:
        dest:
          anyOf:
            - $ref: "#/definitions/substitution_dest"
            - type: array
              minItems: 1
              items:
                $ref: "#/definitions/substitution_dest"
        src:
          type: object
          properties:
            schema:
              type: string
              pattern: ^[A-Za-z]+/[A-Za-z]+/v\d+$
            name:
              type: string
            path:
              type: string
          additionalProperties: false
          required:
            - schema
            - name
            - path
      additionalProperties: false
      required:
```

(continues on next page)

(continued from previous page)

```
          - dest
          - src
    storagePolicy:
      type: string
      enum:
        - encrypted
        - cleartext
  additionalProperties: false
  required:
    - schema
    - name
    - storagePolicy
    - layeringDefinition
```

Validation Schemas

DataSchema schemas validate the `data` section of every document ingested by Deckhand.

All schemas below are `DataSchema` documents. They define additional properties not included in the base schema or override default properties in the base schema.

These schemas are only enforced after validation for the base schema has passed. Failure to pass these schemas will result in an error entry being created for the validation with name `deckhand-schema-validation` corresponding to the created revision.

- `CertificateAuthorityKey` schema.

  JSON schema against which all documents with `deckhand/CertificateAuthorityKey/v1` schema are validated.

Listing 4: Schema for `CertificateAuthorityKey` documents.

```
---
schema: deckhand/DataSchema/v1
metadata:
  name: deckhand/CertificateAuthorityKey/v1
  schema: metadata/Control/v1
data:
  $schema: http://json-schema.org/schema#
  type: string
```

  This schema is used to sanity-check all CertificateAuthorityKey documents that are passed to Deckhand.

- `CertificateAuthority` schema.

  JSON schema against which all documents with `deckhand/CertificateAuthority/v1` schema are validated.

Listing 5: Schema for `CertificateAuthority` documents.

```
---
schema: deckhand/DataSchema/v1
metadata:
  name: deckhand/CertificateAuthority/v1
  schema: metadata/Control/v1
data:
```

(continues on next page)

(continued from previous page)

```
  $schema: http://json-schema.org/schema#
  type: string
```

This schema is used to sanity-check all `CertificateAuthority` documents that are passed to Deckhand.

- `CertificateKey` schema.

  JSON schema against which all documents with `deckhand/CertificateKey/v1` schema are validated.

Listing 6: Schema for `CertificateKey` documents.

```
---
schema: deckhand/DataSchema/v1
metadata:
  name: deckhand/CertificateKey/v1
  schema: metadata/Control/v1
data:
  $schema: http://json-schema.org/schema#
  type: string
```

This schema is used to sanity-check all `CertificateKey` documents that are passed to Deckhand.

- `Certificate` schema.

  JSON schema against which all documents with `deckhand/Certificate/v1` schema are validated.

Listing 7: Schema for `Certificate` documents.

```
---
schema: deckhand/DataSchema/v1
metadata:
  name: deckhand/Certificate/v1
  schema: metadata/Control/v1
data:
  $schema: http://json-schema.org/schema#
  type: string
```

This schema is used to sanity-check all `Certificate` documents that are passed to Deckhand.

- `LayeringPolicy` schema.

  JSON schema against which all documents with `deckhand/LayeringPolicy/v1` schema are validated.

Listing 8: Schema for `LayeringPolicy` documents.

```
---
schema: deckhand/DataSchema/v1
metadata:
  name: deckhand/LayeringPolicy/v1
  schema: metadata/Control/v1
data:
  $schema: http://json-schema.org/schema#
  type: object
  properties:
    layerOrder:
      type: array
      items:
        type: string
  additionalProperties: false
```

(continues on next page)

(continued from previous page)

```
  required:
    - layerOrder
```

This schema is used to sanity-check all `LayeringPolicy` documents that are passed to Deckhand.

- `PrivateKey` schema.

  JSON schema against which all documents with `deckhand/PrivateKey/v1` schema are validated.

Listing 9: Schema for `PrivateKey` documents.

```
---
schema: deckhand/DataSchema/v1
metadata:
  name: deckhand/Passphrase/v1
  schema: metadata/Control/v1
data:
  $schema: http://json-schema.org/schema#
  type: string
```

This schema is used to sanity-check all `PrivateKey` documents that are passed to Deckhand.

- `PublicKey` schema.

  JSON schema against which all documents with `deckhand/PublicKey/v1` schema are validated.

Listing 10: Schema for `PublicKey` documents.

```
---
schema: deckhand/DataSchema/v1
metadata:
  name: deckhand/PublicKey/v1
  schema: metadata/Control/v1
data:
  $schema: http://json-schema.org/schema#
  type: string
```

This schema is used to sanity-check all `PublicKey` documents that are passed to Deckhand.

- `Passphrase` schema.

  JSON schema against which all documents with `deckhand/Passphrase/v1` schema are validated.

Listing 11: Schema for `Passphrase` documents.

```
---
schema: deckhand/DataSchema/v1
metadata:
  name: deckhand/PrivateKey/v1
  schema: metadata/Control/v1
data:
  $schema: http://json-schema.org/schema#
  type: string
```

This schema is used to sanity-check all `Passphrase` documents that are passed to Deckhand.

- `ValidationPolicy` schema.

  JSON schema against which all documents with `deckhand/ValidationPolicy/v1` schema are validated.

Listing 12: Schema for `ValidationPolicy` documents.

```
---
schema: deckhand/DataSchema/v1
metadata:
  name: deckhand/ValidationPolicy/v1
  schema: metadata/Control/v1
data:
  $schema: http://json-schema.org/schema#
  type: object
  properties:
    validations:
      type: array
      items:
        type: object
        properties:
          name:
            type: string
            pattern: ^.*-(validation|verification)$
          expiresAfter:
            type: string
        additionalProperties: false
        required:
          - name
  required:
    - validations
  additionalProperties: false
```

This schema is used to sanity-check all `ValidationPolicy` documents that are passed to Deckhand.

2.1.9 Document Rendering

Document rendering involves extracting all raw revision documents from Deckhand's database, retrieving encrypted information from Barbican, and applying substitution, layering and replacement algorithms on the data.

The following algorithms are involved during the rendering process:

Document Substitution

Substitution provides an "open" data sharing model in which any source document can be used to substitute data into any destination document.

Use Cases

- Sharing of data between specific documents no matter their `schema`.
- Data sharing using pattern matching.
- Fine-grained sharing of specific sections of data.

Document Layering

Layering provides a "restricted" data inheritance model intended to help reduce duplication in configuration.

Use Cases

- Sharing of data between documents with the same `schema`.
- Deep merging of objects and lists.
- Layer order with multiple layers, resulting in a larger hierarchy of documents.
- Source document for data sharing can be identified via labels, allowing for different documents to be used as the source for sharing, depending on *Parent Selection*.

Document Replacement

Replacement builds on top of layering to provide yet another mechanism for reducing data duplication.

Use Cases

- Same as layering, but with a need to replace higher-layer documents with lower-layer documents for specific site deployments.

2.1.10 Document Substitution

Introduction

Document substitution, simply put, allows one document to overwrite *parts* of its own data with that of another document. Substitution involves a source document sharing data with a destination document, which replaces its own data with the shared data.

Substitution may be leveraged as a mechanism for:

- inserting secrets into configuration documents
- reducing data duplication by declaring common data within one document and having multiple other documents substitute data from the common location as needed

During document rendering, substitution is applied at each layer after all merge actions occur. For more information on the interaction between document layering and substitution, see: *Document Rendering*.

Requirements

Substitutions between documents are not restricted by `schema`, `name`, nor `layer`. Source and destination documents do not need to share the same `schema`.

No substitution dependency cycle may exist between a series of substitutions. For example, if A substitutes from B, B from C, and C from A, then Deckhand will raise an exception as it is impossible to determine the source data to use for substitution in the presence of a dependency cycle.

Substitution works like this:

The source document is resolved via the `src.schema` and `src.name` keys and the `src.path` key is used relative to the source document's `data` section to retrieve the substitution data, which is then injected into the `data` section of the destination document using the `dest.path` key.

If all the constraints above are correct, then the substitution source data is injected into the destination document's `data` section, at the path specified by `dest.path`.

The injection of data into the destination document can be more fine-tuned using a regular expression; see the *Substitution with Patterns* section below for more information.

Note: Substitution is only applied to the `data` section of a document. This is because a document's `metadata` and `schema` sections should be immutable within the scope of a revision, for obvious reasons.

Rendering Documents with Substitution

Concrete (non-abstract) documents can be used as a source of substitution into other documents. This substitution is layer-independent, so given the 3 layer example above, which includes `global`, `region` and `site` layers, a document in the `region` layer could insert data from a document in the `site` layer.

Example

Here is a sample set of documents demonstrating substitution:

```
---
schema: deckhand/Certificate/v1
metadata:
  name: example-cert
  storagePolicy: cleartext
  layeringDefinition:
    layer: site
data: |
  CERTIFICATE DATA
---
schema: deckhand/CertificateKey/v1
metadata:
  name: example-key
  storagePolicy: encrypted
  layeringDefinition:
    layer: site
data: |
  KEY DATA
---
schema: deckhand/Passphrase/v1
metadata:
  name: example-password
  storagePolicy: encrypted
  layeringDefinition:
    layer: site
data: my-secret-password
---
schema: armada/Chart/v1
metadata:
  name: example-chart-01
  storagePolicy: cleartext
  layeringDefinition:
    layer: region
  substitutions:
    - dest:
        path: .chart.values.tls.certificate
      src:
```

(continues on next page)

(continued from previous page)

```
        schema: deckhand/Certificate/v1
        name: example-cert
        path: .
    - dest:
        path: .chart.values.tls.key
      src:
        schema: deckhand/CertificateKey/v1
        name: example-key
        path: .
    - dest:
        path: .chart.values.some_url
        pattern: INSERT_[A-Z]+_HERE
      src:
        schema: deckhand/Passphrase/v1
        name: example-password
        path: .
data:
  chart:
    details:
      data: here
    values:
      some_url: http://admin:INSERT_PASSWORD_HERE@service-name:8080/v1
...
```

The rendered document will look like:

```
---
schema: armada/Chart/v1
metadata:
  name: example-chart-01
  storagePolicy: cleartext
  layeringDefinition:
    layer: region
  substitutions:
    - dest:
        path: .chart.values.tls.certificate
      src:
        schema: deckhand/Certificate/v1
        name: example-cert
        path: .
    - dest:
        path: .chart.values.tls.key
      src:
        schema: deckhand/CertificateKey/v1
        name: example-key
        path: .
    - dest:
        path: .chart.values.some_url
        pattern: INSERT_[A-Z]+_HERE
      src:
        schema: deckhand/Passphrase/v1
        name: example-password
        path: .
data:
  chart:
    details:
      data: here
```

(continues on next page)

(continued from previous page)

```
    values:
      some_url: http://admin:my-secret-password@service-name:8080/v1
      tls:
        certificate: |
          CERTIFICATE DATA
        key: |
          KEY DATA
...
```

Substitution with Patterns

Substitution can be controlled in a more fine-tuned fashion using `dest.pattern` (optional) which functions as a regular expression underneath the hood. The `dest.pattern` has the following constraints:

- `dest.path` key must already exist in the `data` section of the destination document and must have an associated value.
- The `dest.pattern` must be a valid regular expression string.
- The `dest.pattern` must be resolvable in the value of `dest.path`.

If the above constraints are met, then more precise substitution via a pattern can be carried out. If `dest.path` is a string or multiline string then all occurrences of `dest.pattern` found in `dest.path` will be replaced. To handle a more complex `dest.path` read *Recursive Replacement of Patterns*.

Example

```
---
# Source document.
schema: deckhand/Passphrase/v1
metadata:
  name: example-password
  schema: metadata/Document/v1
  layeringDefinition:
    layer: site
  storagePolicy: cleartext
data: my-secret-password
---
# Another source document.
schema: deckhand/Passphrase/v1
metadata:
  name: another-password
  schema: metadata/Document/v1
  layeringDefinition:
    layer: site
  storagePolicy: cleartext
data: another-secret-password
---
# Destination document.
schema: armada/Chart/v1
metadata:
  name: example-chart-01
  schema: metadata/Document/v1
  layeringDefinition:
```

(continues on next page)

(continued from previous page)

```
    layer: region
  substitutions:
    - dest:
        path: .chart.values.some_url
        pattern: INSERT_[A-Z]+_HERE
      src:
        schema: deckhand/Passphrase/v1
        name: example-password
        path: .
    - dest:
        path: .chart.values.script
        pattern: INSERT_ANOTHER_PASSWORD
      src:
        schema: deckhand/Passphrase/v1
        name: another-password
        path: .
data:
  chart:
    details:
      data: here
    values:
      some_url: http://admin:INSERT_PASSWORD_HERE@service-name:8080/v1
      script: |
       some_function("INSERT_ANOTHER_PASSWORD")
       another_function("INSERT_ANOTHER_PASSWORD")
```

After document rendering, the output for `example-chart-01` (the destination document) will be:

```
---
schema: armada/Chart/v1
metadata:
  name: example-chart-01
  schema: metadata/Document/v1
  [...]
data:
  chart:
    details:
      data: here
    values:
      # Notice string replacement occurs at exact location specified by
      # ``dest.pattern``.
      some_url: http://admin:my-secret-password@service-name:8080/v1
      script: |
       some_function("another-secret-password")
       another_function("another-secret-password")
```

Recursive Replacement of Patterns

Patterns may also be replaced recursively. This can be achieved by specifying a `pattern` value and `recurse` as `True` (it otherwise defaults to `False`). Best practice is to limit the scope of the recursion as much as possible: e.g. avoid passing in "$" as the `jsonpath`, but rather a JSON path that lives closer to the nested strings in question.

Note: Recursive selection of patterns will only consider matching patterns. Non-matching patterns will be ignored.

Thus, even if recursion can "pass over" non-matching patterns, they will be silently ignored.

```
---
# Source document.
schema: deckhand/Passphrase/v1
metadata:
  name: example-password
  schema: metadata/Document/v1
  layeringDefinition:
    layer: site
  storagePolicy: cleartext
data: my-secret-password
---
# Destination document.
schema: armada/Chart/v1
metadata:
  name: example-chart-01
  schema: metadata/Document/v1
  layeringDefinition:
    layer: region
  substitutions:
    - dest:
        # Note that the path encapsulates all 3 entries that require pattern
        # replacement.
        path: .chart.values
        pattern: INSERT_[A-Z]+_HERE
        recurse:
          # Note that specifying the depth is mandatory. -1 means that all
          # layers are recursed through.
          depth: -1
      src:
        schema: deckhand/Passphrase/v1
        name: example-password
        path: .
data:
  chart:
    details:
      data: here
    values:
      # Notice string replacement occurs for all paths recursively captured
      # by dest.path, since all their patterns match dest.pattern.
      admin_url: http://admin:INSERT_PASSWORD_HERE@service-name:35357/v1
      internal_url: http://internal:INSERT_PASSWORD_HERE@service-name:5000/v1
      public_url: http://public:INSERT_PASSWORD_HERE@service-name:5000/v1
```

After document rendering, the output for `example-chart-01` (the destination document) will be:

```
---
schema: armada/Chart/v1
metadata:
  name: example-chart-01
  schema: metadata/Document/v1
  [...]
data:
  chart:
    details:
      data: here
```

(continues on next page)

(continued from previous page)

```
  values:
    # Notice how the data from the source document is injected into the
    # exact location specified by ``dest.pattern``.
    admin_url: http://admin:my-secret-password@service-name:35357/v1
    internal_url: http://internal:my-secret-passwor@service-name:5000/v1
    public_url: http://public:my-secret-passwor@service-name:5000/v1
```

Note that the recursion depth must be specified. -1 effectively ignores the depth. Any other positive integer will specify how many levels deep to recurse in order to optimize recursive pattern replacement. Take care to specify the required recursion depth or else too-deep patterns won't be replaced.

Substitution of Encrypted Data

Deckhand allows *data to be encrypted using Barbican*. Substitution of encrypted data works the same as substitution of cleartext data.

Note that during the rendering process, source and destination documents receive the secrets stored in Barbican.

2.1.11 Document Layering

Introduction

Layering provides a restricted data inheritance model intended to help reduce duplication in configuration. With layering, child documents can inherit data from parent documents. Through *Layering Actions*, child documents can control exactly what they inherit from their parent. Document layering, conceptually speaking, works much like class inheritance: A child class inherits all variables and methods from its parent, but can elect to override its parent's functionality.

Goals behind layering include:

- model site deployment data hierarchically
- lessen data duplication across site layers (as well as other conceptual layers)

Document Abstraction

Layering works with *Document Abstraction*: child documents can inherit from abstract as well as concrete parent documents.

Pre-Conditions

A document only has one parent, but its parent is computed dynamically using the *Parent Selection* algorithm. That is, the notion of "multiple inheritance" **does not** apply to document layering.

Documents with different `schema` values are never layered together (see the *Document Substitution* section if you need to combine data from multiple types of documents).

Document layering requires a *LayeringPolicy* to exist in the revision whose documents will be layered together (rendered). An error will be issued otherwise.

Terminology

Note: Whether a layer is "lower" or "higher" has entirely to do with its order of initialization in a `layerOrder` and, by extension, its precedence in the *Parent Selection* algorithm described below.

- Layer - A position in a hierarchy used to control *Parent Selection* by the *Algorithm*. It can be likened to a position in an inheritance hierarchy, where `object` in Python can be likened to the highest layer in a `layerOrder` in Deckhand and a leaf class can be likened to the lowest layer in a `layerOrder`.
- Child - Meaningful only in a parent-child document relationship. A document with a lower layer (but higher priority) than its parent, determined using using *Parent Selection*.
- Parent - Meaningful only in a parent-child document relationship. A document with a higher layer (but lower priority) than its child.
- Layering Policy - A *control document* that defines the strict `layerOrder` in which documents are layered together. See *LayeringPolicy* documentation for more information.
- Layer Order (`layerOrder`) - Corresponds to the `data.layerOrder` of the *LayeringPolicy* document. Establishes the layering hierarchy for a set of layers in the system.
- Layering Definition (`layeringDefinition`) - Metadata in each document for controlling the following:
  - `layer`: the document layer itself
  - `parentSelector`: *Parent Selection*
  - `abstract`: *Document Abstraction*
  - `actions`: *Layering Actions*
- Parent Selector (`parentSelector`) - Key-value pairs or labels for identifying the document's parent. Note that these key-value pairs are not unique and that multiple documents can use them. All the key-value pairs in the `parentSelector` must be found among the target parent's `metadata.labels`: this means that the `parentSelector` key-value pairs must be a subset of the target parent's `metadata.labels` key-value pairs. See *Parent Selection* for further details.
- Layering Actions (`actions`) - A list of actions that control what data are inherited from the parent by the child. See *Layering Actions* for further details.

Algorithm

Layering is applied at the bottommost layer of the `layerOrder` first and at the topmost layer of the `layerOrder` last, such that the "base" layers are processed first and the "leaf" layers are processed last. For each layer in the `layerOrder`, the documents that correspond to that layer are retrieved. For each document retrieved, the `layerOrder` hierarchy is resolved using *Parent Selection* to identify the parent document. Finally, the current document is layered with its parent using *Layering Actions*.

After layering is complete, the *Document Substitution* algorithm is applied to the *current* document, if applicable.

Layering Configuration

Layering is configured in 2 places:

1. The `LayeringPolicy` control document (described in *LayeringPolicy*), which defines the valid layers and their order of precedence.

2. In the `metadata.layeringDefinition` section of normal (`metadata.schema=metadata/Document/v1`) documents. For more information about document structure, reference *Document Format*.

An example `layeringDefinition` may look like:

```
layeringDefinition:
  # Controls whether the document is abstract or concrete.
  abstract: true
  # A layer in the ``layerOrder``. Must be valid or else an error is raised.
  layer: region
  # Key-value pairs or labels for identifying the document's parent.
  parentSelector:
    required_key_a: required_label_a
    required_key_b: required_label_b
  # Actions which specify which data to add to the child document.
  actions:
    - method: merge
      path: .path.to.merge.into.parent
    - method: delete
      path: .path.to.delete
```

Layering Actions

Introduction

Layering actions allow child documents to modify data that is inherited from the parent. What if the child document should only inherit some of the parent data? No problem. A merge action can be performed, followed by `delete` and `replace` actions to trim down on what should be inherited.

Each layer action consists of an `action` and a `path`. Whenever *any* action is specified, *all* the parent data is automatically inherited by the child document. The `path` specifies which data from the *child* document to **prioritize over** that of the parent document. Stated differently, all data from the parent is considered while *only* the *child* data at `path` is considered during an action. However, whenever a conflict occurs during an action, the *child* data takes priority over that of the parent.

Layering actions are queued – meaning that if a `merge` is specified before a `replace` then the `merge` will *necessarily* be applied before the `replace`. For example, a `merge` followed by a `replace` **is not necessarily** the same as a `replace` followed by a `merge`.

Layering actions can be applied to primitives, lists and dictionaries alike.

Action Types

Supported actions are:

- `merge` - "deep" merge child data and parent data into the child `data`, at the specified JSONPath

  Note: For conflicts between the child and parent data, the child document's data is **always** prioritized. No other conflict resolution strategy for this action currently exists.

  `merge` behavior depends upon the data types getting merged. For objects and lists, Deckhand uses JSONPath resolution to retrieve data from those entities, after which Deckhand applies merge strategies (see below) to combine merge child and parent data into the child document's `data` section.

  Merge Strategies

Deckhand applies the following merge strategies for each data type:

- object: "Deep-merge" child and parent data together; conflicts are resolved by prioritizing child data over parent data. "Deep-merge" means recursively combining data for each key-value pair in both objects.
- array: The merge strategy involves:
  - When using an index in the action `path` (e.g. `a[0]`):
    1. Copying the parent array into the child's `data` section at the specified JSONPath.
    2. Appending each child entry in the original child array into the parent array. This behavior is synonymous with the `extend` list function in Python.
  - When not using an index in the action `path` (e.g. `a`):
    1. The child's array replaces the parent's array.
- primitives: Includes all other data types, except for `null`. In this case JSONPath resolution is impossible, so child data is prioritized over that of the parent.

Examples

Given:

```
Child Data:    ``{'a': {'x': 7, 'z': 3}, 'b': 4}``
Parent Data:   ``{'a': {'x': 1, 'y': 2}, 'c': 9}``
```

- When:

  ```
  Merge Path: ``.``
  ```

  Then:

  ```
  Rendered Data: ``{'a': {'x': 7, 'y': 2, 'z': 3}, 'b': 4, 'c': 9}``

  All data from parent is automatically considered, all data from child
  is considered due to ``.`` (selects everything), then both merged.
  ```

- When:

  ```
  Merge Path: ``.a``
  ```

  Then:

  ```
  Rendered Data: ``{'a': {'x': 7, 'y': 2, 'z': 3}, 'c': 9}``

  All data from parent is automatically considered, all data from child
  at ``.a`` is considered, then both merged.
  ```

- When:

  ```
  Merge Path: ``.b``
  ```

  Then:

  ```
  Rendered Data: ``{'a': {'x': 1, 'y': 2}, 'b': 4, 'c': 9}``

  All data from parent is automatically considered, all data from child
  at ``.b`` is considered, then both merged.
  ```

- When:

```
Merge Path: ``.c``
```

Then:

```
Error raised (``.c`` missing in child).
```

- `replace` - overwrite existing data with child data at the specified JSONPath.

  Examples

  Given:

```
Child Data:    ``{'a': {'x': 7, 'z': 3}, 'b': 4}``
Parent Data:   ``{'a': {'x': 1, 'y': 2}, 'c': 9}``
```

  – When:

```
Replace Path: ``.``
```

  Then:

```
Rendered Data: ``{'a': {'x': 7, 'z': 3}, 'b': 4}``

All data from parent is automatically considered, but is replaced by all
data from child at ``.`` (selects everything), so replaces everything
in parent.
```

  – When:

```
Replace Path: ``.a``
```

  Then:

```
Rendered Data: ``{'a': {'x': 7, 'z': 3}, 'c': 9}``

All data from parent is automatically considered, but is replaced by all
data from child at ``.a``, so replaces all parent data at ``.a``.
```

  – When:

```
Replace Path: ``.b``
```

  Then:

```
Rendered Data: ``{'a': {'x': 1, 'y': 2}, 'b': 4, 'c': 9}``

All data from parent is automatically considered, but is replaced by all
data from child at ``.b``, so replaces all parent data at ``.b``.

While ``.b`` isn't in the parent, it only needs to exist in the child.
In this case, something (from the child) replaces nothing (from the
parent).
```

  – When:

```
Replace Path: ``.c``
```

  Then:

```
Error raised (``.c`` missing in child).
```

- `delete` - remove the existing data at the specified JSONPath.

  Examples

  Given:

```
Child Data:    ``{'a': {'x': 7, 'z': 3}, 'b': 4}``
Parent Data:   ``{'a': {'x': 1, 'y': 2}, 'c': 9}``
```

  – When:

```
Delete Path: ``.``
```

    Then:

```
Rendered Data: ``{}``

Note that deletion of everything results in an empty dictionary by
default.
```

  – When:

```
Delete Path: ``.a``
```

    Then:

```
Rendered Data: ``{'c': 9}``

All data from Parent Data at ``.a`` was deleted, rest copied over.
```

  – When:

```
Delete Path: ``.c``
```

    Then:

```
Rendered Data: ``{'a': {'x': 1, 'y': 2}}``

All data from Parent Data at ``.c`` was deleted, rest copied over.
```

  – When:

```
Replace Path: ``.b``
```

    Then:

```
Error raised (``.b`` missing in child).
```

After actions are applied for a given layer, substitutions are applied (see the *Document Substitution* section for details).

Parent Selection

Parent selection is performed dynamically. Unlike *Document Substitution*, parent selection does not target a specific document using `schema` and `name` identifiers. Rather, parent selection respects the `layerOrder`, selecting the highest precedence parent in accordance with the algorithm that follows. This allows flexibility in parent selection: if

a document's immediate parent is removed in a revision, then, if applicable, the grandparent (in the previous revision) can become the document's parent (in the latest revision).

Selection of document parents is controlled by the `parentSelector` field and works as follows:

- A given document, `C`, that specifies a `parentSelector`, will have exactly one parent, `P`. If comparing layering with inheritance, layering, then, does *not* allow multi-inheritance.
- Both `C` and `P` must have the **same** `schema`.
- Both `C` and `P` should have **different** `metadata.name` values except in the case of *Document Replacement*.
- Document `P` will be the highest-precedence document whose `metadata.labels` are a **superset** of document C's `parentSelector`. Where:
  - Highest precedence means that `P` belongs to the lowest layer defined in the `layerOrder` list from the `LayeringPolicy` which is at least one level higher than the layer for `C`. For example, if `C` has layer `site`, then its parent `P` must at least have layer `type` or above in the following `layerOrder`:

```
---
...
layerOrder:
  - global # Highest layer
  - type
  - site   # Lowest layer
```

  - Superset means that `P` **at least** has all the labels in its `metadata.labels` that child `C` references via its `parentSelector`. In other words, parent `P` can have more labels than `C` uses to reference it, but `C` must at least have one matching label in its `parentSelector` with `P`.
- Deckhand will select `P` if it belongs to the highest-precedence layer. For example, if `C` belongs to layer `site`, `P` belongs to layer `type`, and `G` belongs to layer `global`, then Deckhand will use `P` as the parent for `C`. If `P` is non-existent, then `G` will be selected instead.

For example, consider the following sample documents:

```
---
schema: deckhand/LayeringPolicy/v1
metadata:
  schema: metadata/Control/v1
  name: layering-policy
data:
  layerOrder:
    - global
    - region
    - site
---
schema: example/Kind/v1
metadata:
  schema: metadata/Document/v1
  name: global-1234
  labels:
    key1: value1
  layeringDefinition:
    abstract: true
    layer: global
data:
  a:
    x: 1
    y: 2
```

(continues on next page)

(continued from previous page)

```
---
schema: example/Kind/v1
metadata:
  schema: metadata/Document/v1
  name: region-1234
  labels:
    key1: value1
  layeringDefinition:
    abstract: true
    layer: region
    parentSelector:
      key1: value1
    actions:
      - method: replace
        path: .a
data:
  a:
    z: 3
---
schema: example/Kind/v1
metadata:
  schema: metadata/Document/v1
  name: site-1234
  layeringDefinition:
    layer: site
    parentSelector:
      key1: value1
    actions:
      - method: merge
        path: .
data:
  b: 4
```

When rendering, the parent chosen for `site-1234` will be `region-1234`, since it is the highest precedence document that matches the label selector defined by `parentSelector`, and the parent chosen for `region-1234` will be `global-1234` for the same reason. The rendered result for `site-1234` would be:

```
---
schema: example/Kind/v1
metadata:
  name: site-1234
data:
  a:
    z: 3
  b: 4
```

If `region-1234` were later removed, then the parent chosen for *site-1234* would become `global-1234`, and the rendered result would become:

```
---
schema: example/Kind/v1
metadata:
  name: site-1234
data:
  a:
    x: 1
```

(continues on next page)

(continued from previous page)

```
    y: 2
  b: 4
```

2.1.12 Document Replacement

Note: Document replacement is an advanced concept in Deckhand. This section assumes that the reader already has an understanding of *Document Layering* and *Document Substitution*.

Document replacement, in the simplest terms, involves a *child* document replacing its *parent*. That is, the *entire* child document replaces its parent document. Replacement aims to lessen data duplication by taking advantage of *Document Abstraction* and document layering patterns.

Unlike the *Document Layering* `replace` action, which allows a child document to selectively replace portions of the parent's `data` section with that of its own, document replacement allows a child document to replace the *entire* parent document.

Todo: Elaborate on these patterns in a separate section.

Replacement introduces the `replacement: true` property underneath the top-level `metadata` section. This property is subject to certain preconditions, discussed in the *Requirements* section below.

Replacement aims to replace specific values in a parent document via document replacement for particular sites, while allowing the same parent document to be consumed directly (layered with, substituted from) for completely different sites. This means that the same YAML template can be referenced from a global namespace by different site-level documents, and when necessary, specific sites can override the global defaults with specific overrides via document replacement. Effectively, this means that the same template can be referenced without having to duplicate all of its data, just to override a few values between the otherwise-exactly-the-same templates.

Like abstract documents, documents that **are replaced** are not returned from Deckhand's `rendered-documents` endpoint. (Documents that **do replace** – those with the `replacement: true` property – are returned instead.)

Requirements

Document replacement has the following requirements:

- Only a child document can replace its parent.
- The child document must have the `replacement: true` property underneath its `metadata` section.
- The child document must be able to select the correct parent. For more information on this, please reference the *Parent Selection* section.
- Additionally, the child document must have the **same** `metadata.name` and `schema` as its parent. Their `metadata.layeringDefinition.layer` must **differ**.

The following result in validation errors:

- A document with `replacement: true` doesn't have a parent.
- A document with `replacement: true` doesn't have the same `metadata.name` and `schema` as its parent.
- A replacement document cannot itself be replaced. That is, only one level of replacement is allowed.

Here are the following possible scenarios regarding child and parent replacement values:

Child	Parent	Status
True	True	Throws InvalidDocumentReplacement exception
False	True	Throws InvalidDocumentReplacement exception
True	False	Valid scenario
False	False	Throws InvalidDocumentReplacement exception

Examples

Note that each key in the examples below is *mandatory* and that the `parentSelector` labels should be able to select the parent to be replaced.

Document **replacer** (child):

```
---
# Note that the schema and metadata.name keys are the same as below.
schema: armada/Chart/v1
metadata:
  name: airship-deckhand
  # The replacement: true key is mandatory.
  replacement: true
  layeringDefinition:
    # Note that the layer differs from that of the parent below.
    layer: N-1
    # The key-value pairs underneath `parentSelector` must be compatible with
    # key-value pairs underneath the `labels` section in the parent document
    # below.
    parentSelector:
      selector: foo
    actions:
      - ...
data: ...
```

Which replaces the document **replacee** (parent):

```
---
# Note that the schema and metadata.name keys are the same as above.
schema: armada/Chart/v1
metadata:
  name: airship-deckhand
  labels:
    selector: foo
  layeringDefinition:
    # Note that the layer differs from that of the child above.
    layer: N
data: ...
```

Why Replacement?

Layering without Replacement

Layering without replacement can introduce a lot of data duplication across documents. Take the following use case: Some sites need to be deployed with log debugging *enabled* and other sites need to be deployed with log debugging *disabled*.

To achieve this, two top-layer documents can be created:

```
---
schema: armada/Chart/v1
metadata:
  name: airship-deckhand-1
  layeringDefinition:
    layer: global
    ...
data:
  debug: false
  # Note that the data below can be arbitrarily long and complex.
  ...
```

And:

```
---
schema: armada/Chart/v1
metadata:
  name: airship-deckhand-2
  layeringDefinition:
    layer: global
    ...
data:
  debug: true
  # Note that the data below can be arbitrarily long and complex.
  ...
```

However, what if the only thing that differs between the two documents is just `debug: true|false` and every other value in both documents is precisely the same?

Clearly, the pattern above leads to a lot of data duplication.

Layering with Replacement

Using document replacement, the above duplication can be partially eliminated. For example:

```
# Replacer (child document).
---
schema: armada/Chart/v1
metadata:
  name: airship-deckhand
  replacement: true
  layeringDefinition:
    layer: site
    parentSelector:
      selector: foo
    actions:
      - method: merge
        path: .
      - method: replace
        path: .debug
data:
  debug: true
  ...
```

And:

```
# Replacee (parent document).
---
schema: armada/Chart/v1
metadata:
  name: airship-deckhand
  labels:
    selector: foo
  layeringDefinition:
    layer: global
    ...
data:
  debug: false
  ...
```

In the case above, for sites that require `debug: false`, only the global-level document should be included in the payload to Deckhand, along with all other documents required for site deployment.

However, for sites that require `debug: true`, both documents should be included in the payload to Deckhand, along with all other documents required for site deployment.

Implications for Pegleg

In practice, when using Pegleg, each document above can be placed in a separate file and Pegleg can either reference *only* the parent document if log debugging needs to be enabled or *both* documents if log debugging needs to be disabled. This pattern allows data duplication to be lessened.

How It Works

Document replacement involves a child document replacing its parent. There are three fundamental cases that are handled:

1. A child document replaces its parent. Only the child is returned.
2. Same as (1), except that the parent document is used as a substitution source. With replacement, the child is used as the substitution source instead.
3. Same as (2), except that the parent document is used as a layering source (that is, yet another child document layers with the parent). With replacement, the child is used as the layering source instead.

2.1.13 Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Operator's Guide

3.1 Operator's Guide

3.1.1 Deckhand API Documentation

API

This API will only support YAML as a serialization format. Since the IETF does not provide an official media type for YAML, this API will use `application/x-yaml`.

This is a description of the `v1.0` API. Documented paths are considered relative to `/api/v1.0`.

PUT `/buckets/{bucket_name}/documents`

Accepts a multi-document YAML body and creates a new revision that updates the contents of the `bucket_name` bucket. Documents from the specified bucket that exist in previous revisions, but are absent from the request are removed from that revision (though still accessible via older revisions).

Documents in other buckets are not changed and will be included in queries for documents of the newly created revision.

Updates are detected based on exact match to an existing document of `schema` + `metadata.name`. It is an error that responds with `409 Conflict` to attempt to PUT a document with the same `schema` + `metadata.name` as an existing document from a different bucket in the most-recent revision.

This endpoint is the only way to add, update, and delete documents. This triggers Deckhand's internal schema validations for all documents.

If no changes are detected, a new revision should not be created. This allows services to periodically re-register their schemas without creating unnecessary revisions.

GET /revisions/{revision_id}/documents

Returns a multi-document YAML response containing all the documents matching the filters specified via query string parameters. Returned documents will be as originally added with no substitutions or layering applied.

Supported query string parameters:

- `schema` - string, optional - The top-level `schema` field to select. This may be partially specified by section, e.g., `schema=promenade` would select all `kind` and `version` schemas owned by promenade, or `schema=promenade/Node` which would select all versions of `promenade/Node` documents. One may not partially specify the namespace or kind, so `schema=promenade/No` would not select `promenade/Node/v1` documents, and `schema=prom` would not select `promenade` documents.
- `metadata.name` - string, optional
- `metadata.layeringDefinition.abstract` - string, optional - Valid values are the "true" and "false".
- `metadata.layeringDefinition.layer` - string, optional - Only return documents from the specified layer.
- `metadata.label` - string, optional, repeatable - Uses the format `metadata.label=key=value`. Repeating this parameter indicates all requested labels must apply (AND not OR).
- `status.bucket` - string, optional, repeatable - Used to select documents only from a particular bucket. Repeating this parameter indicates documents from any of the specified buckets should be returned.
- `sort` - string, optional, repeatable - Defines the sort order for returning results. Default is by creation date. Repeating this parameter indicates use of multi-column sort with the most significant sorting column applied first.
- `order` - string, optional - Valid values are "asc" and "desc". Default is "asc". Controls the order in which the `sort` result is returned: "asc" returns sorted results in ascending order, while "desc" returns results in descending order.
- **`limit` - int, optional - Controls number of documents returned by this** endpoint.
- **`cleartext-secrets` - boolean, optional - Determines if data and substitutions** paths should be redacted (sha256) if a user has access to encrypted files. Default is to redact the values.

GET /revisions/{revision_id}/rendered-documents

Returns a multi-document YAML of fully layered and substituted documents. No abstract documents will be returned. This is the primary endpoint that consumers will interact with for their configuration.

Valid query parameters are the same as for `/revisions/{revision_id}/documents`, minus the parameters in `metadata.layeringDefinition`, which are not supported.

Raises a `409 Conflict` if a `layeringPolicy` document could not be found.

Raises a `500 Internal Server Error` if rendered documents fail schema validation.

GET /revisions

Lists existing revisions and reports basic details including a summary of validation status for each `deckhand/ValidationPolicy` that is part of that revision.

Supported query string parameters:

- `tag` - string, optional, repeatable - Used to select revisions that have been tagged with particular tags.

- `sort` - string, optional, repeatable - Defines the sort order for returning results. Default is by creation date. Repeating this parameter indicates use of multi-column sort with the most significant sorting column applied first.
- `order` - string, optional - Valid values are "asc" and "desc". Default is "asc". Controls the order in which the `sort` result is returned: "asc" returns sorted results in ascending order, while "desc" returns results in descending order.

Sample response:

```
---
count: 7
next: https://deckhand/api/v1.0/revisions?limit=2&offset=2
prev: null
results:
  - id: 1
    url: https://deckhand/api/v1.0/revisions/1
    createdAt: 2017-07-14T21:23Z
    buckets: [mop]
    tags:
      a: {}
    validationPolicies:
      site-deploy-validation:
        status: failure
  - id: 2
    url: https://deckhand/api/v1.0/revisions/2
    createdAt: 2017-07-16T01:15Z
    buckets: [flop, mop]
    tags:
      b:
        random: stuff
        foo: bar
    validationPolicies:
      site-deploy-validation:
        status: success
```

DELETE `/revisions`

Permanently delete all documents.

> **Warning:** This removes all revisions and resets the data store.

GET `/revisions/{{revision_id}}`

Get a detailed description of a particular revision. The status of each `ValidationPolicy` belonging to the revision is also included. Valid values for the status of each validation policy are:

- `success` - All validations associated with the policy are `success`.
- `failure` - Any validation associated with the policy has status `failure`, `expired` or `missing`.

Sample response:

```
---
id: 1
url: https://deckhand/api/v1.0/revisions/1
createdAt: 2017-07-14T021:23Z
buckets: [mop]
tags:
  a:
    random: stuff
    url: https://deckhand/api/v1.0/revisions/1/tags/a
validationPolicies:
  site-deploy-validation:
    url: https://deckhand/api/v1.0/revisions/1/documents?schema=deckhand/
↪ValidationPolicy/v1&name=site-deploy-validation
    status: failure
    validations:
      - name: deckhand-schema-validation
        url: https://deckhand/api/v1.0/revisions/1/validations/deckhand-schema-
↪validation/entries/0
        status: success
      - name: drydock-site-validation
        status: missing
      - name: promenade-site-validation
        url: https://deckhand/api/v1.0/revisions/1/validations/promenade-site-
↪validation/entries/0
        status: expired
      - name: armada-deployability-validation
        url: https://deckhand/api/v1.0/revisions/1/validations/armada-deployability-
↪validation/entries/0
        status: failure
```

Validation status is always for the most recent entry for a given validation. A status of `missing` indicates that no entries have been created. A status of `expired` indicates that the validation had succeeded, but the `expiresAfter` limit specified in the `ValidationPolicy` has been exceeded.

GET /revisions/{{revision_id}}/diff/{{comparison_revision_id}}

This endpoint provides a basic comparison of revisions in terms of how the buckets involved have changed. Only buckets with existing documents in either of the two revisions in question will be reported; buckets with documents that are only present in revisions between the two being compared are omitted from this report. That is, buckets with documents that were accidentally created (and then deleted to rectify the mistake) that are not directly present in the two revisions being compared are omitted.

The response will contain a status of `created`, `deleted`, `modified`, or `unmodified` for each bucket.

The ordering of the two revision ids is not important.

For the purposes of diffing, the `revision_id` "0" is treated as a revision with no documents, so queries comparing revision "0" to any other revision will report "created" for each bucket in the compared revision.

Diffing a revision against itself will respond with the each of the buckets in the revision as `unmodified`.

Diffing revision "0" against itself results in an empty dictionary as the response.

Examples

A response for a typical case, `GET /api/v1.0/revisions/6/diff/3` (or equivalently `GET /api/v1.0/revisions/3/diff/6`).

```
---
bucket_a: created
bucket_b: deleted
bucket_c: modified
bucket_d: unmodified
```

A response for diffing against an empty revision, `GET /api/v1.0/revisions/0/diff/6`:

```
---
bucket_a: created
bucket_c: created
bucket_d: created
```

A response for diffing a revision against itself, `GET /api/v1.0/revisions/6/diff/6`:

```
---
bucket_a: unmodified
bucket_c: unmodified
bucket_d: unmodified
```

Diffing two revisions that contain the same documents, `GET /api/v1.0/revisions/8/diff/11`:

```
---
bucket_e: unmodified
bucket_f: unmodified
bucket_d: unmodified
```

Diffing revision zero with itself, `GET /api/v1.0/revisions/0/diff/0`:

```
---
{}
```

GET /revisions/{{revision_id}}/deepdiff/{{comparison_revision_id}}

This is an advanced version of `diff` api. It provides deepdiff between two revisions of modified buckets.

The response will contain `modified`, `added`, `deleted` documents deepdiff details. Modified documents diff will consist of data and metadata change details. In case the document storagePolicy is encrypted, deepdiff will hide data and will return only `{'encrypted': True}`.

Examples

A response for a typical case, `GET /api/v1.0/revisions/3/deepdiff/4`

```
---
bucket_a: created
bucket_b: deleted
bucket_c: modified
bucket_c diff:
```

(continues on next page)

(continued from previous page)

```
  document_changed:
    count: 1
    details:
      ('example/Kind/v1', 'doc-b'):
        data_changed:
          values_changed:
            root['foo']: {new_value: 3, old_value: 2}
        metadata_changed: {}
```

Document added deepdiff response, `GET /api/v1.0/revisions/4/deepdiff/5`

```
---
bucket_a: created
bucket_c: modified
bucket_c diff:
  document_added:
    count: 1
    details:
    - [example/Kind/v1, doc-c]
```

Document deleted deepdiff response, `GET /api/v1.0/revisions/5/deepdiff/6`

```
---
bucket_a: created
bucket_c: modified
bucket_c diff:
  document_deleted:
    count: 1
    details:
    - [example/Kind/v1, doc-c]
```

A response for deepdiffing against an empty revision, `GET /api/v1.0/revisions/0/deepdiff/2`:

```
---
bucket_a: created
bucket_b: created
```

A response for deepdiffing a revision against itself, `GET /api/v1.0/revisions/6/deepdiff/6`:

```
---
bucket_a: unmodified
bucket_c: unmodified
bucket_d: unmodified
```

DeepDiffing two revisions that contain the same documents, `GET /api/v1.0/revisions/1/deepdiff/2`:

```
---
bucket_a: unmodified
bucket_b: unmodified
```

DeepDiffing revision zero with itself, `GET /api/v1.0/revisions/0/deepdiff/0`:

```
---
{}
```

POST /revisions/{{revision_id}}/validations/{{name}}

Add the results of a validation for a particular revision.

An example `POST` request body indicating validation success:

```
---
status: success
validator:
  name: promenade
  version: 1.1.2
```

An example `POST` request indicating validation failure:

```
POST /api/v1.0/revisions/3/validations/promenade-site-validation
Content-Type: application/x-yaml

---
status: failure
errors:
  - documents:
      - schema: promenade/Node/v1
        name: node-document-name
      - schema: promenade/Masters/v1
        name: kubernetes-masters
    message: Node has master role, but not included in cluster masters list.
validator:
  name: promenade
  version: 1.1.2
```

GET /revisions/{{revision_id}}/validations

Gets the list of validations which have been reported for this revision.

Sample response:

```
---
count: 2
next: null
prev: null
results:
  - name: deckhand-schema-validation
    url: https://deckhand/api/v1.0/revisions/4/validations/deckhand-schema-validation
    status: success
  - name: promenade-site-validation
    url: https://deckhand/api/v1.0/revisions/4/validations/promenade-site-validation
    status: failure
```

GET /revisions/{{revision_id}}/validations/detail

Gets the list of validations, with details, which have been reported for this revision.

Sample response:

```
---
count: 1
next: null
prev: null
results:
  - name: promenade-site-validation
    url: https://deckhand/api/v1.0/revisions/4/validations/promenade-site-validation/
↪entries/0
    status: failure
    createdAt: 2017-07-16T02:03Z
    expiresAfter: null
    expiresAt: null
    errors:
      - documents:
          - schema: promenade/Node/v1
            name: node-document-name
          - schema: promenade/Masters/v1
            name: kubernetes-masters
        message: Node has master role, but not included in cluster masters list.
```

GET /revisions/{{revision_id}}/validations/{{name}}

Gets the list of validation entry summaries that have been posted.

Sample response:

```
---
count: 1
next: null
prev: null
results:
  - id: 0
    url: https://deckhand/api/v1.0/revisions/4/validations/promenade-site-validation/
↪entries/0
    status: failure
```

GET /revisions/{{revision_id}}/validations/{{name}}/entries/{{entry_id}}

Gets the full details of a particular validation entry, including all posted error details.

Sample response:

```
---
name: promenade-site-validation
url: https://deckhand/api/v1.0/revisions/4/validations/promenade-site-validation/
↪entries/0
status: failure
createdAt: 2017-07-16T02:03Z
expiresAfter: null
expiresAt: null
errors:
  - documents:
      - schema: promenade/Node/v1
        name: node-document-name
```

(continues on next page)

(continued from previous page)

```
      - schema: promenade/Masters/v1
        name: kubernetes-masters
    message: Node has master role, but not included in cluster masters list.
```

POST /revisions/{{revision_id}}/tags/{{tag}}

Associate the revision with a collection of metadata, if provided, by way of a tag. The tag itself can be used to label the revision. If a tag by name `tag` already exists, the tag's associated metadata is updated.

Sample request with body:

```
POST ``/revisions/0615b731-7f3e-478d-8ba8-a223eab4757e/tags/foobar``
Content-Type: application/x-yaml

---
thing: bar
```

Sample response:

```
Content-Type: application/x-yaml
HTTP/1.1 201 Created
Location: https://deckhand/api/v1.0/revisions/0615b731-7f3e-478d-8ba8-a223eab4757e/
↪tags/foobar

---
tag: foobar
data:
  thing: bar
```

Sample request without body:

```
POST ``/revisions/0615b731-7f3e-478d-8ba8-a223eab4757e/tags/foobar``
Content-Type: application/x-yaml
```

Sample response:

```
Content-Type: application/x-yaml
HTTP/1.1 201 Created
Location: https://deckhand/api/v1.0/revisions/0615b731-7f3e-478d-8ba8-a223eab4757e/
↪tags/foobar

---
tag: foobar
data: {}
```

GET /revisions/{{revision_id}}/tags

List the tags associated with a revision.

Sample request with body:

```
GET ``/revisions/0615b731-7f3e-478d-8ba8-a223eab4757e/tags``
```

Sample response:

```
Content-Type: application/x-yaml
HTTP/1.1 200 OK

---
- tag: foo
  data:
    thing: bar
- tag: baz
  data:
    thing: qux
```

GET /revisions/{{revision_id}}/tags/{{tag}}

Show tag details for tag associated with a revision.

Sample request with body:

```
GET ``/revisions/0615b731-7f3e-478d-8ba8-a223eab4757e/tags/foo``
```

Sample response:

```
Content-Type: application/x-yaml
HTTP/1.1 200 OK

---
tag: foo
data:
  thing: bar
```

DELETE /revisions/{{revision_id}}/tags/{{tag}}

Delete tag associated with a revision.

Sample request with body:

```
GET ``/revisions/0615b731-7f3e-478d-8ba8-a223eab4757e/tags/foo``
```

Sample response:

```
Content-Type: application/x-yaml
HTTP/1.1 204 No Content
```

DELETE /revisions/{{revision_id}}/tags

Delete all tags associated with a revision.

Sample request with body:

```
GET ``/revisions/0615b731-7f3e-478d-8ba8-a223eab4757e/tags``
```

Sample response:

```
Content-Type: application/x-yaml
HTTP/1.1 204 No Content
```

POST /rollback/{target_revision_id}

Creates a new revision that contains exactly the same set of documents as the revision specified by `target_revision_id`.

3.1.2 Deckhand API Client Library Documentation

The recommended approach to instantiate the Deckhand client is via a Keystone session:

```
from keystoneauth1.identity import v3
from keystoneauth1 import session

keystone_auth = {
    'project_domain_name': PROJECT_DOMAIN_NAME,
    'project_name': PROJECT_NAME,
    'user_domain_name': USER_DOMAIN_NAME,
    'password': PASSWORD,
    'username': USERNAME,
    'auth_url': AUTH_URL,
}
auth = v3.Password(**keystone_auth)
sess = session.Session(auth=auth)
deckhandclient = client.Client(session=sess)
```

You can also instantiate the client via one of Keystone's supported `auth` plugins:

```
from keystoneauth1.identity import v3

keystone_auth = {
    'auth_url': AUTH_URL,
    'token': TOKEN,
    'project_id': PROJECT_ID,
    'project_domain_name': PROJECT_DOMAIN_NAME
}
auth = v3.Token(**keystone_auth)
deckhandclient = client.Client(auth=auth)
```

Which will allow you to authenticate using a pre-existing, project-scoped token.

Alternatively, you can use non-session authentication to instantiate the client, though this approach has been deprecated.

```
from deckhand.client import client

deckhandclient = client.Client(
    username=USERNAME,
    password=PASSWORD,
    project_name=PROECT_NAME,
    project_domain_name=PROJECT_DOMAIN_NAME,
    user_domain_name=USER_DOMAIN_NAME,
    auth_url=AUTH_URL)
```

Note: The Deckhand client by default expects that the service be registered under the Keystone service catalog as `deckhand`. To provide a different value pass `service_type=SERVICE_TYPE` to the `Client` constructor.

After you have instantiated an instance of the Deckhand client, you can invoke the client managers' functionality:

```
# Generate a sample document.
payload = """
---
schema: deckhand/Certificate/v1
metadata:
  schema: metadata/Document/v1
  name: application-api
  storagePolicy: cleartext
data: |-
  -----BEGIN CERTIFICATE-----
  MIIDYDCCAkigAwIBAgIUKG41PW4VtiphzASAMY4/3hL8OtAwDQYJKoZIhvcNAQEL
  ...snip...
  P3WT9CfFARnsw2nKjnglQcwKkKLYip0WY2wh3FE7nrQZP6xKNaSRlh6p2pCGwwwH
  HkvVwA==
  -----END CERTIFICATE-----
"""

# Create a bucket and associate it with the document.
result = client.buckets.update('mop', payload)

>>> result
<Bucket name: mop>

# Convert the response to a dictionary.
>>> result.to_dict()
{'status': {'bucket': 'mop', 'revision': 1},
 'schema': 'deckhand/Certificate/v1', 'data': {...} 'id': 1,
 'metadata': {'layeringDefinition': {'abstract': False},
 'storagePolicy': 'cleartext', 'name': 'application-api',
 'schema': 'metadata/Document/v1'}}

# Show the revision that was created.
revision = client.revisions.get(1)

>>> revision.to_dict()
{'status': 'success', 'tags': {},
 'url': 'https://deckhand/api/v1.0/revisions/1',
 'buckets': ['mop'], 'validationPolicies': [], 'id': 1,
 'createdAt': '2017-12-09T00:15:04.309071'}

# List all revisions.
revisions = client.revisions.list()

>>> revisions.to_dict()
{'count': 1, 'results': [{'buckets': ['mop'], 'id': 1,
 'createdAt': '2017-12-09T00:29:34.031460', 'tags': []}]}

# List raw documents for the created revision.
raw_documents = client.revisions.documents(1, rendered=False)

>>> [r.to_dict() for r in raw_documents]
```

(continues on next page)

(continued from previous page)

```
[{'status': {'bucket': 'foo', 'revision': 1},
  'schema': 'deckhand/Certificate/v1', 'data': {...}, 'id': 1,
  'metadata': {'layeringDefinition': {'abstract': False},
  'storagePolicy': 'cleartext', 'name': 'application-api',
  'schema': 'metadata/Document/v1'}}]
```

Client Reference

For more information about how to use the Deckhand client, refer to the client module.

3.1.3 Deckhand Configuration

Cache Configuration

Deckhand currently uses 3 different caches for the following use cases:

- Caching rendered documents (see *Document Rendering*) for faster future look-ups
- Caching Barbican secret payloads
- Caching `jsonschema` results for quickly resolving deeply nested dictionary data

All 3 caches are implemented in memory.

Please reference the configuration groups below to enable or customize the timeout for each cache:

- `[barbican]`
- `[engine]`
- `[jsonschema]`

Sample Configuration File

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

The sample configuration can also be viewed in file form.

```
[DEFAULT]

#
# From oslo.log
#

# If set to true, the logging level will be set to DEBUG instead of the default
# INFO level. (boolean value)
# Note: This option can be changed without restarting.
#debug = false

# The name of a logging configuration file. This file is appended to any
# existing logging configuration files. For details about logging configuration
# files, see the Python logging module documentation. Note that when logging
# configuration files are used then all logging configuration is set in the
```

(continues on next page)

(continued from previous page)

```
# configuration file and other logging configuration options are ignored (for
# example, logging_context_format_string). (string value)
# Note: This option can be changed without restarting.
# Deprecated group/name - [DEFAULT]/log_config
#log_config_append = <None>

# Defines the format string for %%(asctime)s in log records. Default:
# %(default)s . This option is ignored if log_config_append is set. (string
# value)
#log_date_format = %Y-%m-%d %H:%M:%S

# (Optional) Name of log file to send logging output to. If no default is set,
# logging will go to stderr as defined by use_stderr. This option is ignored if
# log_config_append is set. (string value)
# Deprecated group/name - [DEFAULT]/logfile
#log_file = <None>

# (Optional) The base directory used for relative log_file  paths. This option
# is ignored if log_config_append is set. (string value)
# Deprecated group/name - [DEFAULT]/logdir
#log_dir = <None>

# Uses logging handler designed to watch file system. When log file is moved or
# removed this handler will open a new log file with specified path
# instantaneously. It makes sense only if log_file option is specified and Linux
# platform is used. This option is ignored if log_config_append is set. (boolean
# value)
#watch_log_file = false

# Use syslog for logging. Existing syslog format is DEPRECATED and will be
# changed later to honor RFC5424. This option is ignored if log_config_append is
# set. (boolean value)
#use_syslog = false

# Enable journald for logging. If running in a systemd environment you may wish
# to enable journal support. Doing so will use the journal native protocol which
# includes structured metadata in addition to log messages.This option is
# ignored if log_config_append is set. (boolean value)
#use_journal = false

# Syslog facility to receive log lines. This option is ignored if
# log_config_append is set. (string value)
#syslog_log_facility = LOG_USER

# Use JSON formatting for logging. This option is ignored if log_config_append
# is set. (boolean value)
#use_json = false

# Log output to standard error. This option is ignored if log_config_append is
# set. (boolean value)
#use_stderr = false

# Format string to use for log messages with context. (string value)
#logging_context_format_string = %(asctime)s.%(msecs)03d %(process)d %(levelname)s
↪%(name)s [%(request_id)s %(user_identity)s] %(instance)s%(message)s

# Format string to use for log messages when context is undefined. (string
```

(continues on next page)

(continued from previous page)

```
# value)
#logging_default_format_string = %(asctime)s.%(msecs)03d %(process)d %(levelname)s
↪%(name)s [-] %(instance)s%(message)s

# Additional data to append to log message when logging level for the message is
# DEBUG. (string value)
#logging_debug_format_suffix = %(funcName)s %(pathname)s:%(lineno)d

# Prefix each line of exception output with this format. (string value)
#logging_exception_prefix = %(asctime)s.%(msecs)03d %(process)d ERROR %(name)s
↪%(instance)s

# Defines the format string for %(user_identity)s that is used in
# logging_context_format_string. (string value)
#logging_user_identity_format = %(user)s %(tenant)s %(domain)s %(user_domain)s
↪%(project_domain)s

# List of package logging levels in logger=LEVEL pairs. This option is ignored
# if log_config_append is set. (list value)
#default_log_levels = amqp=WARN,amqplib=WARN,boto=WARN,qpid=WARN,sqlalchemy=WARN,
↪suds=INFO,oslo.messaging=INFO,oslo_messaging=INFO,iso8601=WARN,requests.packages.
↪urllib3.connectionpool=WARN,urllib3.connectionpool=WARN,websocket=WARN,requests.
↪packages.urllib3.util.retry=WARN,urllib3.util.retry=WARN,keystonemiddleware=WARN,
↪routes.middleware=WARN,stevedore=WARN,taskflow=WARN,keystoneauth=WARN,oslo.
↪cache=INFO,dogpile.core.dogpile=INFO

# Enables or disables publication of error events. (boolean value)
#publish_errors = false

# The format for an instance that is passed with the log message. (string value)
#instance_format = "[instance: %(uuid)s] "

# The format for an instance UUID that is passed with the log message. (string
# value)
#instance_uuid_format = "[instance: %(uuid)s] "

# Interval, number of seconds, of log rate limiting. (integer value)
#rate_limit_interval = 0

# Maximum number of logged messages per rate_limit_interval. (integer value)
#rate_limit_burst = 0

# Log level name used by rate limiting: CRITICAL, ERROR, INFO, WARNING, DEBUG or
# empty string. Logs with level greater or equal to rate_limit_except_level are
# not filtered. An empty string means that all levels are filtered. (string
# value)
#rate_limit_except_level = CRITICAL

# Enables or disables fatal status of deprecations. (boolean value)
#fatal_deprecations = false


[database]

#
# From oslo.db
#
```

(continues on next page)

(continued from previous page)

```
# If True, SQLite uses synchronous mode. (boolean value)
#sqlite_synchronous = true

# The back end to use for the database. (string value)
# Deprecated group/name - [DEFAULT]/db_backend
#backend = sqlalchemy

# The SQLAlchemy connection string to use to connect to the database. (string
# value)
# Deprecated group/name - [DEFAULT]/sql_connection
# Deprecated group/name - [DATABASE]/sql_connection
# Deprecated group/name - [sql]/connection
#connection = <None>

# The SQLAlchemy connection string to use to connect to the slave database.
# (string value)
#slave_connection = <None>

# The SQL mode to be used for MySQL sessions. This option, including the
# default, overrides any server-set SQL mode. To use whatever SQL mode is set by
# the server configuration, set this to no value. Example: mysql_sql_mode=
# (string value)
#mysql_sql_mode = TRADITIONAL

# If True, transparently enables support for handling MySQL Cluster (NDB).
# (boolean value)
#mysql_enable_ndb = false

# Connections which have been present in the connection pool longer than this
# number of seconds will be replaced with a new one the next time they are
# checked out from the pool. (integer value)
# Deprecated group/name - [DATABASE]/idle_timeout
# Deprecated group/name - [database]/idle_timeout
# Deprecated group/name - [DEFAULT]/sql_idle_timeout
# Deprecated group/name - [DATABASE]/sql_idle_timeout
# Deprecated group/name - [sql]/idle_timeout
#connection_recycle_time = 3600

# DEPRECATED: Minimum number of SQL connections to keep open in a pool. (integer
# value)
# Deprecated group/name - [DEFAULT]/sql_min_pool_size
# Deprecated group/name - [DATABASE]/sql_min_pool_size
# This option is deprecated for removal.
# Its value may be silently ignored in the future.
# Reason: The option to set the minimum pool size is not supported by
# sqlalchemy.
#min_pool_size = 1

# Maximum number of SQL connections to keep open in a pool. Setting a value of 0
# indicates no limit. (integer value)
# Deprecated group/name - [DEFAULT]/sql_max_pool_size
# Deprecated group/name - [DATABASE]/sql_max_pool_size
#max_pool_size = 5

# Maximum number of database connection retries during startup. Set to -1 to
# specify an infinite retry count. (integer value)
```

(continues on next page)

(continued from previous page)

```
# Deprecated group/name - [DEFAULT]/sql_max_retries
# Deprecated group/name - [DATABASE]/sql_max_retries
#max_retries = 10

# Interval between retries of opening a SQL connection. (integer value)
# Deprecated group/name - [DEFAULT]/sql_retry_interval
# Deprecated group/name - [DATABASE]/reconnect_interval
#retry_interval = 10

# If set, use this value for max_overflow with SQLAlchemy. (integer value)
# Deprecated group/name - [DEFAULT]/sql_max_overflow
# Deprecated group/name - [DATABASE]/sqlalchemy_max_overflow
#max_overflow = 50

# Verbosity of SQL debugging information: 0=None, 100=Everything. (integer
# value)
# Minimum value: 0
# Maximum value: 100
# Deprecated group/name - [DEFAULT]/sql_connection_debug
#connection_debug = 0

# Add Python stack traces to SQL as comment strings. (boolean value)
# Deprecated group/name - [DEFAULT]/sql_connection_trace
#connection_trace = false

# If set, use this value for pool_timeout with SQLAlchemy. (integer value)
# Deprecated group/name - [DATABASE]/sqlalchemy_pool_timeout
#pool_timeout = <None>

# Enable the experimental use of database reconnect on connection lost. (boolean
# value)
#use_db_reconnect = false

# Seconds between retries of a database transaction. (integer value)
#db_retry_interval = 1

# If True, increases the interval between retries of a database operation up to
# db_max_retry_interval. (boolean value)
#db_inc_retry_interval = true

# If db_inc_retry_interval is set, the maximum seconds between retries of a
# database operation. (integer value)
#db_max_retry_interval = 10

# Maximum retries in case of connection error or deadlock error before error is
# raised. Set to -1 to specify an infinite retry count. (integer value)
#db_max_retries = 20

# Optional URL parameters to append onto the connection URL at connect time;
# specify as param1=value1&param2=value2&... (string value)
#connection_parameters =


[oslo_policy]

#
# From oslo.policy
```

(continues on next page)

(continued from previous page)

```
#

# This option controls whether or not to enforce scope when evaluating policies.
# If ``True``, the scope of the token used in the request is compared to the
# ``scope_types`` of the policy being enforced. If the scopes do not match, an
# ``InvalidScope`` exception will be raised. If ``False``, a message will be
# logged informing operators that policies are being invoked with mismatching
# scope. (boolean value)
#enforce_scope = false

# This option controls whether or not to use old deprecated defaults when
# evaluating policies. If ``True``, the old deprecated defaults are not going to
# be evaluated. This means if any existing token is allowed for old defaults but
# is disallowed for new defaults, it will be disallowed. It is encouraged to
# enable this flag along with the ``enforce_scope`` flag so that you can get the
# benefits of new defaults and ``scope_type`` together (boolean value)
#enforce_new_defaults = false

# The relative or absolute path of a file that maps roles to permissions for a
# given service. Relative paths must be specified in relation to the
# configuration file setting this option. (string value)
#policy_file = policy.json

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

# Directories where policy configuration files are stored. They can be relative
# to any directory in the search path defined by the config_dir option, or
# absolute paths. The file defined by policy_file must exist for these
# directories to be searched.  Missing or empty directories are ignored. (multi
# valued)
#policy_dirs = policy.d

# Content Type to send and receive data for REST based policy check (string
# value)
# Possible values:
# application/x-www-form-urlencoded - <No description provided>
# application/json - <No description provided>
#remote_content_type = application/x-www-form-urlencoded

# server identity verification for REST based policy check (boolean value)
#remote_ssl_verify_server_crt = false

# Absolute path to ca cert file for REST based policy check (string value)
#remote_ssl_ca_crt_file = <None>

# Absolute path to client cert for REST based policy check (string value)
#remote_ssl_client_crt_file = <None>

# Absolute path client key file REST based policy check (string value)
#remote_ssl_client_key_file = <None>
```

3.1.4 Deckhand Exceptions

For a list of Deckhand exceptions as well as debugging information related to each please reference Deckhand's errors module.

3.1.5 Multiple Distro Support

This project builds images for Deckhand component only. Currently, it supports building images for ubuntu and opensuse (leap 15.1 as base image).

By default, Ubuntu images are built and are published to public registry server. Recently support for publishing opensuse image has been added.

If you need to build opensuse images locally, the following parameters can be passed to the *make* command in deckhand repository's root directory with *images* as target:

```
DISTRO: opensuse_15
DISTRO_BASE_IMAGE: "opensuse/leap:15.1"
DOCKER_REGISTRY: { your_docker_registry }
IMAGE_TAG: latest
PUSH_IMAGE: false
```

Following is an example in command format to build and publish images locally. Command is run in deckhand repository's root directory.

> make images DISTRO=opensuse_15 DOCKER_REGISTRY={ your_docker_registry} IMAGE_TAG=latest PUSH_IMAGE=true

Following parameters need to be passed as environment/shell variable to make command:

DISTRO parameter to identify distro specific Dockerfile, ubuntu_xenial (Default)

DISTRO_BASE_IMAGE parameter to use different base image other than what's used in DISTRO specific Dockerfile (optional)

DOCKER_REGISTRY parameter to specify local/internal docker registry if need to publish image (optional), quay.io (Default)

IMAGE_TAG tag to be used for image built, untagged (Default)

PUSH_IMAGE flag to indicate if images needs to be pushed to a docker registry, false (Default)

This work is done as per approved spec multi_distro_support. Currently only image building logic is enhanced to support multiple distro.

Adding New Distro Support

To add support for building images for a new distro, following steps can be followed.

1. Distro specific deckhand image can be built and tested locally first.
2. Add distro specific Dockerfile which will have steps to include necessary packages and run environment configuration. Use existing Dockerfile as sample to identify needed packages and environment information.
3. New dockerfile can be named as Dockefile.{DISTRO} where DISTRO is expected to be distro identifier which is passed to makefile.
4. Respective dockerfile needs to be placed in {deckahnd_repo}/images/deckhand/
5. Add check, gate, and post jobs for building, testing and publishing images. These entries need to be added in {deckhand_repo}/.zuul.yaml file. You may refer to existing zuul file to opensuse support to understand its usage pattern.
6. Add any relevant information to this document.

3.1.6 Indices and tables

- genindex
- modindex
- search

CHAPTER 4

Contrbitutor's Guide

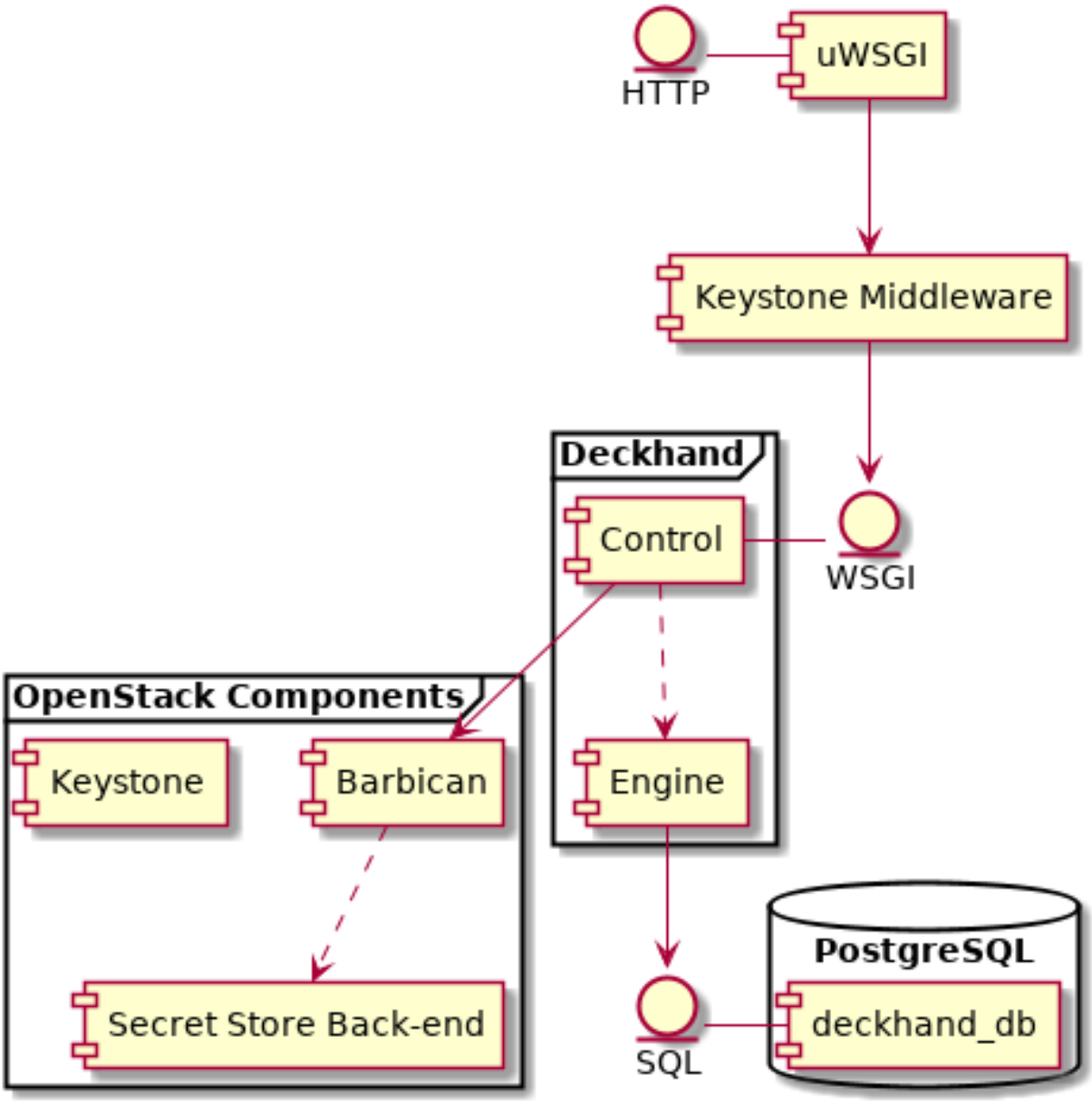
4.1 Contributor's Guide

4.1.1 Contributor Overview

Developer Overview of Deckhand

The core objective of Deckhand is to provide storage, rendering, validation and version control for declarative YAML documents. Deckhand ingests raw, Airship-formatted documents and outputs fully rendered documents to other Airship components.

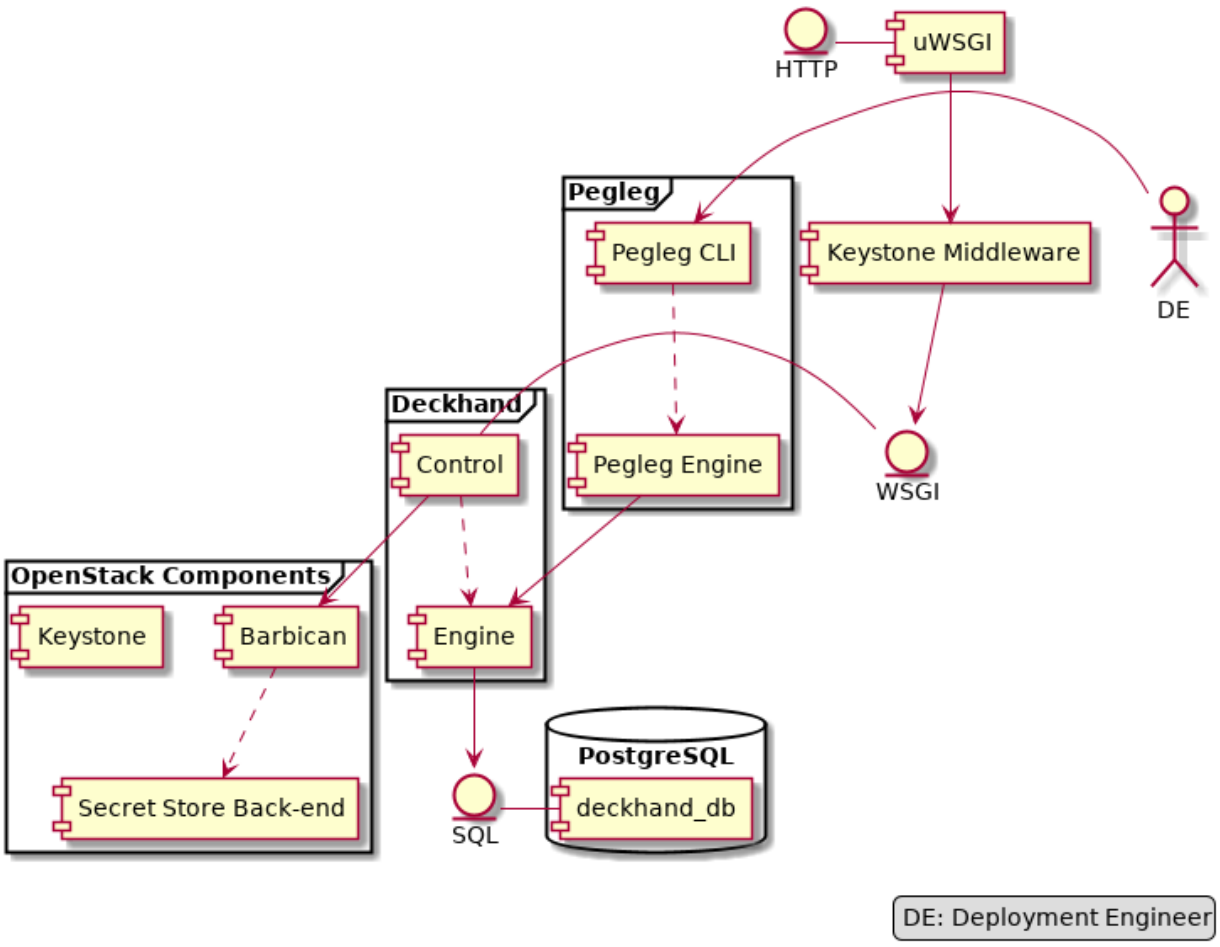
Architecture



From a high-level perspective, Deckhand consists of a RESTful API, a document rendering engine, and a PostgreSQL relational database for document storage. Deckhand ingests Airship-formatted documents, validates them, and stores them in its database for future processing. On demand, Deckhand will fully render the documents, after which they can be consumed by the other Airship components.

Deckhand uses Barbican to securely storage sensitive document data.

Pegleg in effect provides Deckhand with a CLI, which facilitates communication with Deckhand.



Components

control

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

The `control` module is also responsible for communicating with Barbican, which it uses to store and retrieve document *secrets*, which it passes to the `engine` module for *Document Rendering*.

engine

The `engine` module is the interface responsible for all *Document Rendering*. Rendering consists of applying a series of algorithms to the documents, including: topological sorting, *Document Layering*, *Document Substitution*, and *Document Replacement*. This module also realizes revision-diffing and revision-deepdiffing functionality.

db

The `db` module is responsible for implementing the database tables needed to store all Airship documents. This module also realizes version control.

client

The API client library provides an interface for other services to communicate with Deckhand's API. Requires Keystone authentication to use.

Developer Workflow

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

Deckhand strives to conform to the Airship coding conventions.

Python

The Deckhand code base lives under `/deckhand`. Deckhand supports py35 through py37 versions of interpreters.

See *Deckhand Coding Guide* for more information on contribution guidelines.

Docker

The distribution specific Deckhand Dockerfile.{DISTRO} is located in `/images/deckhand` along with any artifacts built specifically to enable the container image. Make targets are used for generating and testing the artifacts.

- `make images` - Build the Deckhand Docker image.

Helm

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

- `make charts` - Pull down dependencies for the Deckhand charts and package everything into a `.tgz` file.
- `make helm_lint` - Lint the Helm charts.
- `make dry-run` - Render the chart and output the Kubernetes manifest YAML documents.

Testing

All Deckhand tests are nested under `/deckhand/tests`.

Deckhand comes equipped with a number of tox targets for running unit and functional tests. See *Development Utilities* for a list of commands.

See *Testing* for more information on testing guidelines.

4.1.2 Contribution Guidelines

Deckhand Coding Guide

Deckhand Style Commandments

- Step 1: Read the OpenStack Style Commandments https://docs.openstack.org/hacking/latest/
- Step 2: Read on

Deckhand Specific Commandments

- [D316] Change assertTrue(isinstance(A, B)) by optimal assert like assertIsInstance(A, B).
- [D317] Change assertEqual(type(A), B) by optimal assert like assertIsInstance(A, B).
- [D320] Setting CONF.* attributes directly in tests is forbidden.
- [D322] Method's default argument shouldn't be mutable.
- [D324] Ensure that jsonutils.%(fun)s must be used instead of json.%(fun)s
- [D325] str() and unicode() cannot be used on an exception. Remove use or use six.text_type()
- [D334] Change assertTrue/False(A in/not in B, message) to the more specific assertIn/NotIn(A, B, message)
- [D335] Check for usage of deprecated assertRaisesRegexp
- [D336] Must use a dict comprehension instead of a dict constructor with a sequence of key-value pairs.
- [D338] Change assertEqual(A in B, True), assertEqual(True, A in B), assertEqual(A in B, False) or assertEqual(False, A in B) to the more specific assertIn/NotIn(A, B)
- [D339] Check common raise_feature_not_supported() is used for v2.1 HTTPNotImplemented response.
- [D344] Python 3: do not use dict.iteritems.
- [D345] Python 3: do not use dict.iterkeys.
- [D346] Python 3: do not use dict.itervalues.
- [D350] Policy registration should be in the central location `deckhand/policies/`.
- [D352] LOG.warn is deprecated. Enforce use of LOG.warning.
- [D355] Enforce use of assertTrue/assertFalse
- [D356] Enforce use of assertIs/assertIsNot
- [D357] Use oslo_utils.uuidutils or uuidsentinel(in case of test cases) to generate UUID instead of uuid4().
- [D358] Return must always be followed by a space when returning a value.

Creating Unit Tests

For every new feature, unit tests should be created that both test and (implicitly) document the usage of said feature. If submitting a patch for a bug that had no unit test, a new passing unit test should be added. If a submitted bug fix does have a unit test, be sure to add a new one that fails without the patch and passes with the patch.

Running Tests

The testing system is based on a combination of tox and testr. The canonical approach to running tests is to simply run the command `tox`. This will create virtual environments, populate them with dependencies and run all of the tests that OpenStack CI systems run. Behind the scenes, tox is running `testr run --parallel`, but is set up such that you can supply any additional testr arguments that are needed to tox. For example, you can run: `tox --analyze-isolation` to cause tox to tell testr to add –analyze-isolation to its argument list.

Functional testing leverages gabbi and requires docker as a prerequisite to be run. Functional tests can be executing by running the command `tox -e functional`.

Building Docs

Normal Sphinx docs can be built via the setuptools `build_sphinx` command. To do this via `tox`, simply run `tox -e docs`, which will cause a virtualenv with all of the needed dependencies to be created and then inside of the virtualenv, the docs will be created and put into doc/build/html.

Reviewing Deckhand Code

Reviewing Deckhand Code

To start read the OpenStack Common Review Checklist

Unit Tests

For any change that adds new functionality to either common functionality or fixes a bug unit tests are required. This is to ensure we don't introduce future regressions and to test conditions which we may not hit in the gate runs.

Functional Tests

For any change that adds major new functionality functional tests are required. This is to ensure that the Deckhand API follows the contract it promises. In addition, functional tests are run against the Deckhand container, which uses an image built from the latest source code to validate the integrity of the image.

Deprecated Code

Deprecated code should go through a deprecation cycle – long enough for other Airship projects to modify their code base to reference new code. Features, APIs or configuration options are marked deprecated in the code. Appropriate warnings will be sent to the end user, operator or library user.

When to approve

- Every patch needs two +2s before being approved.
- Its OK to hold off on an approval until a subject matter expert reviews it.
- If a patch has already been approved but requires a trivial rebase to merge, you do not have to wait for a second +2, since the patch has already had two +2s.

4.1.3 Other Resources

Rest API Policy Enforcement

Policy enforcement in Deckhand leverages the `oslo.policy` library like all OpenStack projects. The implementation is located in `deckhand.policy`. Two types of policy authorization exist in Deckhand:

1) Decorator-level authorization used for wrapping around `falcon` "on_{HTTP_VERB}" methods. In this case, if policy authorization fails a 403 Forbidden is always raised.
2) Conditional authorization, which means that the policy is only enforced if a certain set of conditions are true.

Deckhand, for example, will only conditionally enforce listing encrypted documents if a document's `metadata.storagePolicy` is "encrypted".

Policy Implementation

Deckhand uses `authorize` from `oslo.policy` as the latter supports both `enforce` and `authorize`. `authorize` is stricter because it'll raise an exception if the policy action is not registered under `deckhand.policies` (which enumerates all the legal policy actions and their default rules). This means that attempting to enforce anything not found in `deckhand.policies` will error out with a 'Policy not registered' message.

See Deckhand's policy module for more details.

Sample Policy File

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

The sample configuration can also be viewed in file form.

```
```

Testing

Note: Deckhand has only been tested against a Ubuntu 16.04 environment. The guide below assumes the user is using Ubuntu.

Unit testing

Prerequisites

pifpaf is used to spin up a temporary postgresql database for unit tests. The DB URL is set up as an environment variable via `PIFPAF_URL` which is referenced by Deckhand's unit test suite.

1. PostgreSQL must be installed. To do so, run:

```
$ sudo apt-get update
$ sudo apt-get install postgresql postgresql-contrib -y
```

2. When running `pifpaf run postgresql` (implicitly called by unit tests below), pifpaf uses `pg_config` which can be installed by running:

```
$ sudo apt-get install libpq-dev -y
```

Overview

Unit testing currently uses an in-memory SQLite database. Since Deckhand's primary function is to serve as the back-end storage for Airship, the majority of unit tests perform actual database operations. Mocking is used sparingly because Deckhand is a fairly insular application that lives at the bottom of a very deep stack; Deckhand only communicates with Keystone and Barbican. As such, validating database operations is paramount to correctly testing Deckhand.

To run unit tests using SQLite, execute:

```
$ tox -epy35
```

against a py35-backed environment, respectively.

To run unit tests using PostgreSQL, execute:

```
$ tox -epy35-postgresql
```

To run individual unit tests, run (for example):

```
$ tox -e py35 -- deckhand.tests.unit.db.test_revisions
```

Warning: It is **not** recommended to run postgresql-backed unit tests concurrently. Only run them serially. This is because, to guarantee true test isolation, the DB tables are re-created each test run. Only one instance of PostgreSQL is created across all threads, thus causing major conflicts if concurrency > 1.

Functional testing

Prerequisites

- Docker

  Deckhand requires Docker to run its functional tests. A basic installation guide for Docker for Ubuntu can be found here

- uwsgi

  Can be installed on Ubuntu systems via:

```
sudo apt-get install uwsgi -y
```

Overview

Deckhand uses gabbi as its functional testing framework. Functional tests can be executed via:

```
$ tox -e functional-dev
```

You can also run a subset of tests via a regex:

```
$ tox -e functional-dev -- gabbi.suitemaker.test_gabbi_document-crud-success-multi-
↪bucket
```

The command executes `tools/functional-tests.sh` which:

1) Launches Postgresql inside a Docker container.
2) Sets up a basic Deckhand configuration file that uses Postgresql in its `oslo_db` connection string.
3) Sets up a custom policy file with very liberal permissions so that gabbi can talk to Deckhand without having to authenticate against Keystone and pass an admin token to Deckhand.
4) Instantiates Deckhand via `uwisgi`.
5) Calls gabbi which runs a battery of functional tests.
6) An HTML report that visualizes the result of the test run is output to `results/index.html`.

Note that functional tests can be run concurrently; the flags `--workers` and `--threads` which are passed to `uwsgi` can be > 1.

Todo: At this time, there are no functional tests for policy enforcement verification. Negative tests will be added at a later date to confirm that a 403 Forbidden is raised for each endpoint that does policy enforcement absent necessary permissions.

CICD

Since it is important to validate the Deckhand image itself, CICD:

- Generates the Deckhand image from the new patchset
- Runs functional tests against the just-produced Deckhand image

Deckhand uses the same script – `tools/functional-tests.sh` – for CICD testing. To test Deckhand against a containerized image, run, for example:

```
export DECKHAND_IMAGE=quay.io/airshipit/deckhand:latest-ubuntu_bionic
tox -e functional-dev
```

Which will result in the following script output:

```
Running Deckhand via Docker
+ sleep 5
+ sudo docker run --rm --net=host -p 9000:9000 -v /opt/stack/deckhand/tmp.oBJ6XScFgC:/
↪etc/deckhand quay.io/airshipit/deckhand:latest-ubuntu_bionic
```

Warning: For testing dev changes, it is **not** recommended to follow this approach, as the most up-to-date code is located in the repository itself. Running tests against a remote image will likely result in false positives.

Troubleshooting

- For any errors related to `tox`:

Ensure that `tox` is installed:

```
$ sudo apt-get install tox -y
```

- For any errors related to running `tox -e py35`:

  Ensure that `python3-dev` is installed:

```
$ sudo apt-get install python3-dev -y
```

deckhand

deckhand package

Subpackages

deckhand.barbican package

Submodules

deckhand.barbican.cache module

deckhand.barbican.cache.**invalidate**()

deckhand.barbican.cache.**lookup_by_payload**(*barbicanclient*, ***kwargs*)

Look up secret reference using the secret payload.

Allows for quick lookup of secret references using `secret_payload` via caching (essentially a reverse-lookup).

Useful for ensuring that documents with the same secret payload (which occurs when the same document is recreated across different revisions) persist the same secret reference in the database – and thus quicker future `lookup_by_ref` lookups.

deckhand.barbican.cache.**lookup_by_ref**(*barbicanclient*, *secret_ref*)

Look up secret object using secret reference.

Allows for quick lookup of secret payloads using `secret_ref` via caching.

deckhand.barbican.client_wrapper module

class deckhand.barbican.client_wrapper.**BarbicanClientWrapper**

Bases: `object`

Barbican client wrapper class that encapsulates authentication logic.

call(*method*, **args*, ***kwargs*)

Call a barbican client method and retry on stale token.

Parameters

- **method** – Name of the client method to call as a string.
- **args** – Client method arguments.
- **kwargs** – Client method keyword arguments.

- **retry_on_conflict** – Boolean value. Whether the request should be retried in case of a conflict error (HTTP 409) or not. If retry_on_conflict is False the cached instance of the client won't be used. Defaults to True.

deckhand.barbican.driver module

class deckhand.barbican.driver.**BarbicanDriver**
Bases: object

create_secret(*secret_doc*)
Create a secret.

Parameters **secret_doc** (document.DocumentDict) – Document with storagePolicy of "encrypted".

Returns Secret reference returned by Barbican

Return type str

delete_secret(*secret_ref*)
Delete a secret.

get_secret(*secret_ref*, *src_doc*)
Get a secret.

Module contents

deckhand.client package

Submodules

deckhand.client.base module

Base utilities to build API operation managers and objects on top of.

class deckhand.client.base.**Manager**(*api*)
Bases: object

Manager for API service.

Managers interact with a particular type of API (buckets, revisions, etc.) and provide CRUD operations for them.

api_version

client

resource_class = None

class deckhand.client.base.**Resource**(*manager*, *info*, *loaded=False*)
Bases: object

Base class for OpenStack resources (tenant, user, etc.).

This is pretty much just a bag for attributes.

HUMAN_ID = False

NAME_ATTR = 'name'

api_version

get()
Support for lazy loading details.

Some clients, such as novaclient have the option to lazy load the details, details which can be loaded with this function.

human_id
Human-readable ID which can be used for bash completion.

is_loaded()

set_info(*key*, *value*)

set_loaded(*val*)

to_dict()

deckhand.client.base.**get_url_with_filter**(*url*, *filters*)

deckhand.client.base.**getid**(*obj*)
Get object's ID or object.

Abstracts the common pattern of allowing both an object or an object's ID as a parameter when dealing with relationships.

deckhand.client.base.**prepare_query_string**(*params*)
Convert dict params to query string

deckhand.client.buckets module

class deckhand.client.buckets.**Bucket**(*manager*, *info*, *loaded=False*)
Bases: *deckhand.client.base.Resource*

class deckhand.client.buckets.**BucketManager**(*api*)
Bases: *deckhand.client.base.Manager*

Manage *Bucket* resources.

resource_class
alias of *Bucket*

update(*bucket_name*, *documents*)
Create, update or delete documents associated with a bucket.

Parameters

- **bucket_name** (*str*) – Gets or creates a bucket by this name.
- **documents** (*str*) – YAML-formatted string of Deckhand-compatible documents to create in the bucket.

Returns The created documents along with their associated bucket and revision.

deckhand.client.client module

Deckhand Client interface. Handles the REST calls and responses.

class deckhand.client.client.**Client**(*api_version=None*, *auth=None*, *auth_token=None*, *auth_url=None*, *cacert=None*, *cert=None*, *direct_use=True*, *endpoint_override=None*, *endpoint_type='publicURL'*, *http_log_debug=False*, *insecure=False*, *logger=None*, *password=None*, *project_domain_id=None*, *project_domain_name=None*, *project_id=None*, *project_name=None*, *region_name=None*, *service_name=None*, *service_type='deckhand'*, *session=None*, *timeout=None*, *user_domain_id=None*, *user_domain_name=None*, *user_id=None*, *username=None*, ***kwargs*)

Bases: object

Top-level object to access the Deckhand API.

api_version

projectid

tenant_id

class deckhand.client.client.**SessionClient**(**args*, ***kwargs*)

Bases: keystoneauth1.adapter.Adapter

Wrapper around keystoneauth1 client session implementation and used internally by *Client* below.

Injects Deckhand-specific YAML headers necessary for communication with the Deckhand API.

client_name = 'python-deckhandclient'

client_version = '1.0'

request(*url*, *method*, ***kwargs*)

deckhand.client.exceptions module

Exception definitions.

exception deckhand.client.exceptions.**BadRequest**(*code*, *url*, *method*, *message=None*, *details=None*, *reason=None*, *apiVersion=None*, *retry=False*, *status=None*, *kind=None*, *metadata=None*)

Bases: *deckhand.client.exceptions.ClientException*

HTTP 400 - Bad request: you sent some malformed data.

http_status = 400

message = 'Bad request'

exception deckhand.client.exceptions.**ClientException**(*code*, *url*, *method*, *message=None*, *details=None*, *reason=None*, *apiVersion=None*, *retry=False*, *status=None*, *kind=None*, *metadata=None*)

Bases: Exception

The base exception class for all exceptions this library raises.

message = 'Unknown Error'

exception deckhand.client.exceptions.**Conflict**(*code*, *url*, *method*, *message=None*, *details=None*, *reason=None*, *apiVersion=None*, *retry=False*, *status=None*, *kind=None*, *metadata=None*)

Bases: *deckhand.client.exceptions.ClientException*

HTTP 409 - Conflict

http_status = 409

message = 'Conflict'

exception deckhand.client.exceptions.**Forbidden**(*code*, *url*, *method*, *message=None*, *details=None*, *reason=None*, *apiVersion=None*, *retry=False*, *status=None*, *kind=None*, *metadata=None*)

Bases: *deckhand.client.exceptions.ClientException*

HTTP 403 - Forbidden: your credentials don't give you access to this resource.

http_status = 403

message = 'Forbidden'

exception deckhand.client.exceptions.**HTTPNotImplemented**(*code*, *url*, *method*, *message=None*, *details=None*, *reason=None*, *apiVersion=None*, *retry=False*, *status=None*, *kind=None*, *metadata=None*)

Bases: *deckhand.client.exceptions.ClientException*

HTTP 501 - Not Implemented: the server does not support this operation.

http_status = 501

message = 'Not Implemented'

exception deckhand.client.exceptions.**MethodNotAllowed**(*code*, *url*, *method*, *message=None*, *details=None*, *reason=None*, *apiVersion=None*, *retry=False*, *status=None*, *kind=None*, *metadata=None*)

Bases: *deckhand.client.exceptions.ClientException*

HTTP 405 - Method Not Allowed

http_status = 405

message = 'Method Not Allowed'

exception deckhand.client.exceptions.**NotFound**(*code*, *url*, *method*, *message=None*, *details=None*, *reason=None*, *apiVersion=None*, *retry=False*, *status=None*, *kind=None*, *metadata=None*)

Bases: *deckhand.client.exceptions.ClientException*

HTTP 404 - Not found

http_status = 404

message = 'Not found'

exception deckhand.client.exceptions.**Unauthorized**(*code*, *url*, *method*, *message=None*, *details=None*, *reason=None*, *apiVersion=None*, *retry=False*, *status=None*, *kind=None*, *metadata=None*)

Bases: *deckhand.client.exceptions.ClientException*

HTTP 401 - Unauthorized: bad credentials.

http_status = 401

message = 'Unauthorized'

deckhand.client.exceptions.**from_response**(*response*, *body*, *url*, *method=None*)

Return an instance of a ClientException or subclass based on a request's response.

deckhand.client.revisions module

class deckhand.client.revisions.**Revision**(*manager*, *info*, *loaded=False*)

Bases: *deckhand.client.base.Resource*

class deckhand.client.revisions.**RevisionManager**(*api*)

Bases: *deckhand.client.base.Manager*

Manage *Revision* resources.

deepdiff(*revision_id*, *comparison_revision_id*)

Get revision deepdiff between two revisions.

delete_all()

Delete all revisions.

> **Warning:** Effectively the same as purging the entire database.

diff(*revision_id*, *comparison_revision_id*)

Get revision diff between two revisions.

documents(*revision_id*, *rendered=True*, ***filters*)

Get a list of revision documents or rendered documents.

Parameters

- **revision_id** (*int*) – Revision ID.
- **rendered** (*bool*) – If True, returns list of rendered documents. Else returns list of unmodified, raw documents.
- **filters** – Filters to apply to response body.

Returns List of documents or rendered documents.

Return type list[*Revision*]

get(*revision_id*)

Get details for a revision.

list(***filters*)

Get a list of revisions.

resource_class

alias of *Revision*

rollback(*revision_id*)
Rollback to a previous revision, effectively creating a new one.

deckhand.client.tags module

class deckhand.client.tags.**RevisionTag**(*manager*, *info*, *loaded=False*)
Bases: *deckhand.client.base.Resource*

class deckhand.client.tags.**RevisionTagManager**(*api*)
Bases: *deckhand.client.base.Manager*

Manage *RevisionTag* resources.

create(*revision_id*, *tag*, *data=None*)
Create a revision tag.

delete(*revision_id*, *tag*)
Delete a revision tag.

delete_all(*revision_id*)
Delete all revision tags.

get(*revision_id*, *tag*)
Get details for a revision tag.

list(*revision_id*)
Get list of revision tags.

resource_class
alias of *RevisionTag*

Module contents

deckhand.common package

Submodules

deckhand.common.document module

class deckhand.common.document.**DocumentDict**
Bases: dict

Wrapper for a document.

Implements convenient properties for nested, commonly accessed document keys. Property setters are only implemented for mutable data.

Useful for accessing nested dictionary keys without having to worry about exceptions getting thrown.

Note: As a rule of thumb, setters for any metadata properties should be avoided. Only implement or use for well-understood edge cases.

---

actions

data

classmethod from_list(*documents*)
Convert an iterable of documents into instances of this class.

Parameters **documents** (*iterable*) – Documents to wrap in this class.

has_barbican_ref

has_replacement

is_abstract

is_control

is_encrypted

is_replacement

labels

layer

layer_order

layeringDefinition

layering_definition

meta

metadata

name

parent_selector

classmethod redact(*field*)

replaced_by

schema

storage_policy

substitutions

deckhand.common.document.**document_dict_representer**(*dumper*, *data*)

deckhand.common.utils module

deckhand.common.utils.**deepfilter**(*dct*, ***filters*)
Match `dct` against all the filters in `filters`.

Check whether `dct` matches all the fitlers in `filters`. The filters can reference nested attributes, attributes that are contained within other dictionaries within `dct`.

Useful for querying whether `metadata.name` or `metadata.layeringDefinition.layerOrder` match specific values.

Parameters

- **dct** (*dict*) – The dictionary to check against all the `filters`.
- **filters** (*dict*) – Dictionary of key-value pairs used for filtering out unwanted results.

Returns True if the dictionary satisfies all the filters, else False.

deckhand.common.utils.**jsonpath_parse**(*data*, *jsonpath*, *match_all=False*)

Parse value in the data for the given `jsonpath`.

Retrieve the nested entry corresponding to `data[jsonpath]`. For example, a `jsonpath` of ".foo.bar.baz" means that the data section should conform to:

```
---
foo:
    bar:
        baz: <data_to_be_extracted_here>
```

Parameters

- **data** – The *data* section of a document.
- **jsonpath** – A multi-part key that references a nested path in `data`.
- **match_all** – Whether to return all matches or just the first one.

Returns Entry that corresponds to `data[jsonpath]` if present, else None.

Example:

```
src_name = sub['src']['name']
src_path = sub['src']['path']
src_doc = db_api.document_get(schema=src_schema, name=src_name)
src_secret = utils.jsonpath_parse(src_doc['data'], src_path)
# Do something with the extracted secret from the source document.
```

deckhand.common.utils.**jsonpath_replace**(*data*, *value*, *jsonpath*, *pattern=None*, *recurse=None*)

Update value in `data` at the path specified by `jsonpath`.

If the nested path corresponding to `jsonpath` isn't found in `data`, the path is created as an empty `{}` for each sub-path along the `jsonpath`.

Example:

```
doc = {
    'data': {
        'some_url': http://admin:INSERT_PASSWORD_HERE@svc-name:8080/v1
    }
}
secret = 'super-duper-secret'
path = '$.some_url'
pattern = 'INSERT_[A-Z]+_HERE'
replaced_data = utils.jsonpath_replace(
    doc['data'], secret, path, pattern)
# The returned URL will look like:
# http://admin:super-duper-secret@svc-name:8080/v1
doc['data'].update(replaced_data)
```

Parameters

- **data** – The `data` section of a document.
- **value** – The new value for `data[jsonpath]`.
- **jsonpath** – A multi-part key that references a nested path in `data`. Must begin with "." or "$" (without quotes).

- **pattern** – A regular expression pattern.
- **recurse** – Dictionary containing a single key called "depth" which specifies the recursion depth. If provided, indicates that recursive pattern substitution should be performed, beginning at `jsonpath`. Best practice is to limit the scope of the recursion as much as possible: e.g. avoid passing in "$" as the `jsonpath`, but rather a JSON path that lives closer to the nested strings in question. Optimize performance by choosing an ideal `depth` value; -1 will cause recursion depth to be infinite.

Returns Updated value at `data[jsonpath]`.

Raises MissingDocumentPattern if `pattern` is not None and `data[jsonpath]` doesn't exist.

Raises **ValueError** – If `jsonpath` doesn't begin with "."

deckhand.common.utils.**multisort**(*data*, *sort_by=None*, *order_by=None*)

Sort a dictionary by multiple keys.

The order of the keys is important. The first key takes precedence over the second key, and so forth.

Parameters

- **data** – Dictionary to be sorted.
- **sort_by** (*list or string*) – list or string of keys to sort `data` by.

Returns Sorted dictionary by each key.

deckhand.common.utils.**redact_document**(*document*)

Redact `data` and `substitutions` sections for `document`.

Parameters **document** (*dict*) – Document whose data to redact.

Returns Document with redacted data.

Return type dict

deckhand.common.utils.**redact_documents**(*documents*)

Redact sensitive data for each document in `documents`.

Sensitive data includes `data`, `substitutions[n].src.path`, and `substitutions[n].dest.path` fields.

Parameters **documents** (*list[dict]*) – List of documents whose data to redact.

Returns Documents with redacted sensitive data.

Return type list[dict]

deckhand.common.utils.**to_camel_case**(*s*)

Convert string to camel case.

deckhand.common.utils.**to_snake_case**(*name*)

Convert string to snake case.

deckhand.common.validation_message module

class deckhand.common.validation_message.**ValidationMessage**(*message='Document validation error.', error=True, name='Deckhand validation error', level='Error', doc_schema='', doc_name='', doc_layer='', diagnostic=''*)

Bases: object

ValidationMessage per Airship convention: https://github.com/openstack/airship-in-a-bottle/blob/master/doc/source/api-conventions.rst#output-structure # noqa

Construction of ValidationMessage message:

Parameters

- **message** (*string*) – Validation failure message.
- **error** (*boolean*) – True or False, if this is an error message.
- **name** (*string*) – Identifying name of the validation.
- **level** (*string*) – The severity of validation result, as "Error", "Warning", or "Info"
- **schema** (*string*) – The schema of the document being validated.
- **doc_name** (*string*) – The name of the document being validated.
- **diagnostic** (*string*) – Information about what lead to the message, or details for resolution.

format_message()

Return ValidationMessage message.

Returns The ValidationMessage for the Validation API response.

Return type dict

Module contents

deckhand.conf package

Submodules

deckhand.conf.config module

deckhand.conf.config.**list_opts**()

deckhand.conf.config.**register_opts**(*conf*)

deckhand.conf.opts module

deckhand.conf.opts.**list_opts**()
Entry point used only in the context of sample file generation.

This is the single point of entry to generate the sample configuration file for Deckhand. It collects all the necessary info from the other modules in this package. It is assumed that:

- every other module in this package has a 'list_opts' function which return a dict where * the keys are strings which are the group names * the value of each key is a list of config options for that group
- the deckhand.conf package doesn't have further packages with config options

Module contents

deckhand.control package

Subpackages

deckhand.control.views package

Submodules

deckhand.control.views.document module

class deckhand.control.views.document.**ViewBuilder**
Bases: *deckhand.control.common.ViewBuilder*

Model document API responses as a python dictionary.

There are 2 cases for rendering the response body below.

1. Treat the case where all documents in a bucket have been deleted as a special case. The response body must still include the revision_id and bucket_id. It is not meaningful to include other data about the deleted documents as technically they don't exist. 2. Add all non-deleted documents to the response body.

list(*documents*)

deckhand.control.views.revision module

class deckhand.control.views.revision.**ViewBuilder**
Bases: *deckhand.control.common.ViewBuilder*

Model revision API responses as a python dictionary.

list(*revisions*)

show(*revision*)
Generate view for showing revision details.

Each revision's documents should only be validation policies.

deckhand.control.views.revision_tag module

class deckhand.control.views.revision_tag.**ViewBuilder**
Bases: *deckhand.control.common.ViewBuilder*

Model revision tag API responses as a python dictionary.

list(*tags*)

show(*tag*)

deckhand.control.views.validation module

class deckhand.control.views.validation.**ViewBuilder**
Bases: *deckhand.control.common.ViewBuilder*

Model validation API responses as a python dictionary.

detail(*entries*)

list(*validations*)

list_entries(*entries*)

show(*validation*)

show_entry(*entry*)

Module contents

Submodules

deckhand.control.api module

deckhand.control.api.**init_application**()
Main entry point for initializing the Deckhand API service.

Create routes for the v1.0 API and sets up logging.

deckhand.control.api.**setup_logging**(*conf*)

deckhand.control.base module

class deckhand.control.base.**BaseResource**
Bases: object

Base resource class for implementing API resources.

from_yaml(*req*, *expect_list=True*, *allow_empty=False*)
Reads and converts YAML-formatted request body into a dict or list of dicts.

Parameters

- **req** – Falcon Request object.
- **expect_list** – Whether to expect a list or an object.
- **allow_empty** – Whether the request body can be empty.

Returns List of dicts if `expect_list` is True or else a dict.

no_authentication_methods = []

on_options(*req*, *resp*)

class deckhand.control.base.**DeckhandRequest**(*env*, *options=None*)
Bases: `falcon.request.Request`

context_type
alias of *deckhand.context.RequestContext*

project_id

roles

user_id

deckhand.control.buckets module

class deckhand.control.buckets.**BucketsResource**
Bases: *deckhand.control.base.BaseResource*

API resource for realizing CRUD operations for buckets.

on_put(*req*, *resp*, *bucket_name=None*)

view_builder = <deckhand.control.views.document.ViewBuilder object>

deckhand.control.common module

class deckhand.control.common.**ViewBuilder**
Bases: `object`

Model API responses as dictionaries.

deckhand.control.common.**get_rendered_docs**(*revision_id*, *cleartext_secrets=False*, ***filters*)
Helper for retrieving rendered documents for `revision_id`.

Retrieves raw documents from DB, renders them, and returns rendered result set.

Parameters

- **revision_id** (*int*) – Revision ID whose documents to render.
- **cleartext_secrets** (*bool*) – Whether to show unencrypted data as cleartext.
- **filters** – Filters used for retrieving raw documents from DB.

Returns List of rendered documents.

Return type list[dict]

deckhand.control.common.**invalidate_cache_data**()
Invalidate all data associated with document rendering.

deckhand.control.common.**sanitize_params**(*allowed_params*)
Sanitize query string parameters passed to an HTTP request.

Overrides the `params` attribute in the `req` object with the sanitized params. Invalid parameters are ignored.

Parameters **allowed_params** – The request's query string parameters.

deckhand.control.health module

class deckhand.control.health.**HealthResource**
Bases: *deckhand.control.base.BaseResource*

Basic health check for Deckhand

A resource that allows other Airship components to access and validate Deckhand's health status. The response must be returned within 30 seconds for Deckhand to be deemed "healthy". Unauthenticated GET.

no_authentication_methods = ['GET']

on_get(*req*, *resp*)

deckhand.control.middleware module

class deckhand.control.middleware.**ContextMiddleware**
Bases: object

process_resource(*req*, *resp*, *resource*, *params*)
Handle the authentication needs of the routed request.

Parameters

- **req** – falcon request object that will be examined for method
- **resource** – falcon resource class that will be examined for authentication needs by looking at the no_authentication_methods list of http methods. By default, this will assume that all requests need authentication unless noted in this array. Note that this does not bypass any authorization checks, which will fail if the user is not authenticated.

Raises falcon.HTTPUnauthorized: when value of the 'X-Identity-Status' header is not 'Confirmed' and anonymous access is disallowed.

class deckhand.control.middleware.**HookableMiddlewareMixin**
Bases: object

Provides methods to extract before and after hooks from WSGI Middleware Prior to falcon 0.2.0b1, it's necessary to provide falcon with middleware as "hook" functions that are either invoked before (to process requests) or after (to process responses) the API endpoint code runs. This mixin allows the process_request and process_response methods from a typical WSGI middleware object to be extracted for use as these hooks, with the appropriate method signatures.

as_after_hook()
Extract process_response method as "after" hook :return: after hook function

as_before_hook()
Extract process_request method as "before" hook :return: before hook function

class deckhand.control.middleware.**LoggingMiddleware**
Bases: object

process_resource(*req*, *resp*, *resource*, *params*)

process_response(*req*, *resp*, *resource*, *req_succeeded*)

class deckhand.control.middleware.**YAMLTranslator**
Bases: *deckhand.control.middleware.HookableMiddlewareMixin*, object

Middleware for converting all responses (error and success) to YAML.

falcon error exceptions use JSON formatting and headers by default. This middleware will intercept all responses and guarantee they are YAML format.

Note: This does not include the 401 Unauthorized that is raised by keystonemiddleware which is executed in the pipeline before falcon middleware.

---

process_request(*req*, *resp*)
Performs content type enforcement on behalf of REST verbs.

process_response(*req*, *resp*, *resource*)
Converts responses to application/x-yaml content type.

deckhand.control.no_oauth_middleware module

class deckhand.control.no_oauth_middleware.**NoAuthFilter**(*app*, *forged_roles=None*)
Bases: object

PasteDeploy filter for NoAuth to be used in testing.

deckhand.control.no_oauth_middleware.**noauth_filter_factory**(*global_conf*, *forged_roles*)
Create a NoAuth paste deploy filter

Parameters **forged_roles** – A space seperated list for roles to forge on requests

deckhand.control.revision_deepdiffing module

class deckhand.control.revision_deepdiffing.**RevisionDeepDiffingResource**
Bases: *deckhand.control.base.BaseResource*

API resource for realizing revision deepdiffing.

on_get(*req*, *resp*, *revision_id*, *comparison_revision_id*)

deckhand.control.revision_diffing module

class deckhand.control.revision_diffing.**RevisionDiffingResource**
Bases: *deckhand.control.base.BaseResource*

API resource for realizing revision diffing.

on_get(*req*, *resp*, *revision_id*, *comparison_revision_id*)

deckhand.control.revision_documents module

class deckhand.control.revision_documents.**RenderedDocumentsResource**
Bases: *deckhand.control.base.BaseResource*

API resource for realizing rendered documents endpoint.

Rendered documents are also revision documents, but unlike revision documents, they are finalized documents, having undergone secret substitution and document layering.

Returns a multi-document YAML response containing all the documents matching the filters specified via query string parameters. Returned documents will have secrets substituted into them and be layered with other documents in the revision, in accordance with the `LayeringPolicy` that currently exists in the system.

on_get(*req*, *resp*, *revision_id*)

view_builder = <deckhand.control.views.document.ViewBuilder object>

class deckhand.control.revision_documents.**RevisionDocumentsResource**
Bases: *deckhand.control.base.BaseResource*

API resource for realizing revision documents endpoint.

on_get(*req*, *resp*, *revision_id*)
Returns all documents for a *revision_id*.

Returns a multi-document YAML response containing all the documents matching the filters specified via query string parameters. Returned documents will be as originally posted with no substitutions or layering applied.

view_builder = <deckhand.control.views.document.ViewBuilder object>

deckhand.control.revision_tags module

class deckhand.control.revision_tags.**RevisionTagsResource**
Bases: *deckhand.control.base.BaseResource*

API resource for realizing CRUD for revision tags.

on_delete(*req*, *resp*, *revision_id*, *tag=None*)
Deletes a single tag or deletes all tags for a revision.

on_get(*req*, *resp*, *revision_id*, *tag=None*)
Show tag details or list all tags for a revision.

on_post(*req*, *resp*, *revision_id*, *tag=None*)
Creates a revision tag.

deckhand.control.revisions module

class deckhand.control.revisions.**RevisionsResource**
Bases: *deckhand.control.base.BaseResource*

API resource for realizing CRUD operations for revisions.

on_delete(*req*, *resp*)

on_get(*req*, *resp*, *revision_id=None*)
Returns list of existing revisions.

Lists existing revisions and reports basic details including a summary of validation status for each *deckhand/ValidationPolicy* that is part of each revision.

view_builder = <deckhand.control.views.revision.ViewBuilder object>

deckhand.control.rollback module

class deckhand.control.rollback.**RollbackResource**
Bases: *deckhand.control.base.BaseResource*

API resource for realizing revision rollback.

on_post (*req*, *resp*, *revision_id*)

view_builder = <deckhand.control.views.revision.ViewBuilder object>

deckhand.control.validations module

class deckhand.control.validations.**ValidationsDetailsResource**
Bases: *deckhand.control.base.BaseResource*

API resource for listing revision validations with details.

on_get (*req*, *resp*, *revision_id*)

view_builder = <deckhand.control.views.validation.ViewBuilder object>

class deckhand.control.validations.**ValidationsResource**
Bases: *deckhand.control.base.BaseResource*

API resource for realizing validations endpoints.

on_get (*req*, *resp*, *revision_id*, *validation_name=None*, *entry_id=None*)

on_post (*req*, *resp*, *revision_id*, *validation_name*)

view_builder = <deckhand.control.views.validation.ViewBuilder object>

deckhand.control.versions module

class deckhand.control.versions.**VersionsResource**
Bases: *deckhand.control.base.BaseResource*

Versions resource

Returns the list of supported versions of the Deckhand API. Unauthenticated GET.

no_authentication_methods = ['GET']

on_get (*req*, *resp*)

Module contents

deckhand.db package

Subpackages

deckhand.db.sqlalchemy package

Submodules

deckhand.db.sqlalchemy.api module

Defines interface for DB access.

deckhand.db.sqlalchemy.api.**bucket_get_all** (*session=None*, ***filters*)
Return list of all buckets.

Parameters **session** – Database session object.

Returns List of dictionary representations of retrieved buckets.

deckhand.db.sqlalchemy.api.**bucket_get_or_create**(*bucket_name*, *session=None*)
Retrieve or create bucket.

Retrieve the Bucket DB object by bucket_name if it exists or else create a new Bucket DB object by bucket_name.

Parameters

- **bucket_name** – Unique identifier used for creating or retrieving a bucket.
- **session** – Database session object.

Returns Dictionary representation of created/retrieved bucket.

deckhand.db.sqlalchemy.api.**document_delete**(*document*, *revision_id*, *bucket*, *session=None*)
Delete a document

Creates a new document with the bare minimum information about the document that is to be deleted, and then sets the appropriate deleted fields

Parameters

- **document** – document object/dict to be deleted
- **revision_id** – id of the revision where the document is to be deleted
- **bucket** – bucket object/dict where the document will be deleted from
- **session** – Database session object.

Returns dict representation of deleted document

deckhand.db.sqlalchemy.api.**document_get**(*session=None*, *raw_dict=False*, *revision_id=None*, ***filters*)
Retrieve the first document for revision_id that match filters.

Parameters

- **session** – Database session object.
- **raw_dict** – Whether to retrieve the exact way the data is stored in DB if True, else the way users expect the data.
- **revision_id** – The ID corresponding to the Revision object. If the it is "latest", then retrieve the latest revision, if one exists.
- **filters** – Dictionary attributes (including nested) used to filter out revision documents.

Returns Dictionary representation of retrieved document.

Raises DocumentNotFound if the document wasn't found.

deckhand.db.sqlalchemy.api.**document_get_all**(*session=None*, *raw_dict=False*, *revision_id=None*, ***filters*)
Retrieve all documents for revision_id that match filters.

Parameters

- **session** – Database session object.
- **raw_dict** – Whether to retrieve the exact way the data is stored in DB if True, else the way users expect the data.

- **revision_id** – The ID corresponding to the `Revision` object. If the it is "latest", then retrieve the latest revision, if one exists.
- **filters** – Dictionary attributes (including nested) used to filter out revision documents.

Returns Dictionary representation of each retrieved document.

`deckhand.db.sqlalchemy.api.`**`documents_create`**(*bucket_name*, *documents*, *session=None*)

Create a set of documents and associated bucket.

If no changes are detected, a new revision will not be created. This allows services to periodically re-register their schemas without creating unnecessary revisions.

Parameters

- **bucket_name** – The name of the bucket with which to associate created documents.
- **documents** – List of documents to be created.
- **session** – Database session object.

Returns List of created documents in dictionary format.

Raises **DocumentExists** – If the document already exists in the DB for any bucket.

`deckhand.db.sqlalchemy.api.`**`documents_delete_from_buckets_list`**(*bucket_names*, *session=None*)

Delete all documents in the provided list of buckets

Parameters

- **bucket_names** – list of bucket names for which the associated buckets and their documents need to be deleted.
- **session** – Database session object.

Returns A new model.Revisions object after all the documents have been deleted.

`deckhand.db.sqlalchemy.api.`**`drop_db`**()

`deckhand.db.sqlalchemy.api.`**`get_engine`**()

`deckhand.db.sqlalchemy.api.`**`get_session`**(*autocommit=True*, *expire_on_commit=False*)

`deckhand.db.sqlalchemy.api.`**`raw_query`**(*query*, ***kwargs*)

Execute a raw query against the database.

`deckhand.db.sqlalchemy.api.`**`require_revision_exists`**(*f*)

Decorator to require the specified revision to exist.

Requires the wrapped function to use revision_id as the first argument. If revision_id is not provided, then the check is not performed.

`deckhand.db.sqlalchemy.api.`**`require_unique_document_schema`**(*schema=None*)

Decorator to enforce only one singleton document exists in the system.

An example of a singleton document is a `LayeringPolicy` document.

Only one singleton document can exist within the system at any time. It is an error to attempt to insert a new document with the same `schema` if it has a different `metadata.name` than the existing document.

A singleton document that already exists can be updated, if the document that is passed in has the same name/schema as the existing one.

The existing singleton document can be replaced by first deleting it and only then creating a new one.

Raises ***SingletonDocumentConflict*** – if a singleton document in the system already exists and any of the documents to be created has the same `schema` but has a `metadata.name` that differs from the one already registered.

`deckhand.db.sqlalchemy.api.`**`revision_create`**`(`*session=None*`)`
Create a revision.

Parameters **`session`** – Database session object.

Returns Dictionary representation of created revision.

`deckhand.db.sqlalchemy.api.`**`revision_delete_all`**`()`
Delete all revisions and resets primary key index back to 1 for each table in the database.

> **Warning:** Effectively purges all data from database.

Parameters **`session`** – Database session object.

Returns None

`deckhand.db.sqlalchemy.api.`**`revision_documents_get`**`(`*revision_id=None*, *include_history=True*, *unique_only=True*, *session=None*, ***filters*`)`
Return the documents that match filters for the specified *revision_id*.

Parameters

- **`revision_id`** – The ID corresponding to the `Revision` object. If the ID is `None`, then retrieve the latest revision, if one exists.
- **`include_history`** – Return all documents for revision history prior and up to current revision, if `True`. Default is `True`.
- **`unique_only`** – Return only unique documents if `True`. Default is `True`.
- **`session`** – Database session object.
- **`filters`** – Key-value pairs used for filtering out revision documents.

Returns All revision documents for `revision_id` that match the `filters`, including document revision history if applicable.

Raises ***RevisionNotFound*** – if the revision was not found.

`deckhand.db.sqlalchemy.api.`**`revision_get`**`(`*revision_id=None*, *session=None*`)`
Return the specified *revision_id*.

Parameters

- **`revision_id`** – The ID corresponding to the `Revision` object.
- **`session`** – Database session object.

Returns Dictionary representation of retrieved revision.

Raises ***RevisionNotFound*** – if the revision was not found.

`deckhand.db.sqlalchemy.api.`**`revision_get_all`**`(`*session=None*, ***filters*`)`
Return list of all revisions.

Parameters **`session`** – Database session object.

Returns List of dictionary representations of retrieved revisions.

deckhand.db.sqlalchemy.api.**revision_get_latest**(*session=None*)
Return the latest revision.

Parameters **session** – Database session object.

Returns Dictionary representation of latest revision.

deckhand.db.sqlalchemy.api.**revision_rollback**(*revision_id*, *latest_revision*, *session=None*)
Rollback the latest revision to revision specified by `revision_id`.

Rolls back the latest revision to the revision specified by `revision_id` thereby creating a new, carbon-copy revision.

Parameters

- **revision_id** – Revision ID to which to rollback.
- **latest_revision** – Dictionary representation of the latest revision in the system.

Returns The newly created revision.

deckhand.db.sqlalchemy.api.**revision_tag_create**(*revision_id*, *tag*, *data=None*, *session=None*)
Create a revision tag.

If a tag already exists by name `tag`, the request is ignored.

Parameters

- **revision_id** – ID corresponding to `Revision` DB object.
- **tag** – Name of the revision tag.
- **data** – Dictionary of data to be associated with tag.
- **session** – Database session object.

Returns The tag that was created if not already present in the database, else None.

Raises ***RevisionTagBadFormat*** – If data is neither None nor dictionary.

deckhand.db.sqlalchemy.api.**revision_tag_delete**(*revision_id*, *tag*, *session=None*)
Delete a specific tag for a revision.

Parameters

- **revision_id** – ID corresponding to `Revision` DB object.
- **tag** – Name of the revision tag.
- **session** – Database session object.

Returns None

deckhand.db.sqlalchemy.api.**revision_tag_delete_all**(*revision_id*, *session=None*)
Delete all tags for a revision.

Parameters

- **revision_id** – ID corresponding to `Revision` DB object.
- **session** – Database session object.

Returns None

deckhand.db.sqlalchemy.api.**revision_tag_get**(*revision_id*, *tag*, *session=None*)
Retrieve tag details.

Parameters

- **revision_id** – ID corresponding to `Revision` DB object.
- **tag** – Name of the revision tag.
- **session** – Database session object.

Returns None

Raises ***RevisionTagNotFound*** – If `tag` for `revision_id` was not found.

`deckhand.db.sqlalchemy.api.`**revision_tag_get_all**(*revision_id*, *session=None*)

Return list of tags for a revision.

Parameters

- **revision_id** – ID corresponding to `Revision` DB object.
- **tag** – Name of the revision tag.
- **session** – Database session object.

Returns List of tags for `revision_id`, ordered by the tag name by default.

`deckhand.db.sqlalchemy.api.`**setup_db**(*connection_string*, *create_tables=False*)

`deckhand.db.sqlalchemy.api.`**validation_create**(*revision_id*, *val_name*, *val_data*, *session=None*)

`deckhand.db.sqlalchemy.api.`**validation_get_all**(*revision_id*, *session=None*)

`deckhand.db.sqlalchemy.api.`**validation_get_all_entries**(*revision_id*, *val_name=None*, *session=None*)

`deckhand.db.sqlalchemy.api.`**validation_get_entry**(*revision_id*, *val_name*, *entry_id*, *session=None*)

deckhand.db.sqlalchemy.models module

class `deckhand.db.sqlalchemy.models.`**DeckhandBase**

Bases: `oslo_db.sqlalchemy.models.ModelBase`, `oslo_db.sqlalchemy.models.TimestampMixin`

Base class for Deckhand Models.

created_at = Column(None, DateTime(), table=None, nullable=False, default=ColumnDefaul

deleted = Column(None, Boolean(), table=None, nullable=False, default=ColumnDefault(Fa

deleted_at = Column(None, DateTime(), table=None)

items()

Make the model object behave like a dict.

keys()

Make the model object behave like a dict.

safe_delete(*session=None*)

save(*session=None*)

Save this object.

to_dict()

Convert the object into dictionary format.

updated_at = Column(None, DateTime(), table=None, onupdate=ColumnDefault(<function Dec

values()

deckhand.db.sqlalchemy.models.**create_tables**(*engine*)
Creates database tables for all models with the given engine.

This will be done only by tests that do not have their tables set up by Alembic running during the associated helm chart db_sync job.

deckhand.db.sqlalchemy.models.**register_models**(*engine*, *connection_string*)
Register the sqlalchemy tables itno the BASE.metadata

Sets up the database model objects. Does not create the tables in the associated configured database. (see create_tables)

deckhand.db.sqlalchemy.models.**unregister_models**(*engine*)
Drop database tables for all models with the given engine.

Module contents

Module contents

deckhand.engine package

Submodules

deckhand.engine.cache module

deckhand.engine.cache.**invalidate**()
Invalidate the entire cache.

deckhand.engine.cache.**invalidate_one**(*revision_id*)
Invalidate single entry in cache.

Parameters **revision_id** (*int*) – Revision to invalidate.

deckhand.engine.cache.**lookup_by_revision_id**(*revision_id*, *documents*, ***kwargs*)
Look up rendered documents by `revision_id`.

Parameters

- **revision_id** (*int*) – Revision ID for which to render documents. Used as key in cache.
- **documents** (*List[dict]*) – List of raw documents to render.
- **kwargs** – Kwargs to pass to `render`.

Returns Tuple, where first arg is rendered documents and second arg indicates whether cache was hit.

Return type Tuple[dict, boolean]

deckhand.engine.document_validation module

class deckhand.engine.document_validation.**BaseValidator**
Bases: `object`

Abstract base validator.

Sub-classes should override this to implement schema-specific document validation.

validate(*document*)
Validate whether `document` passes schema validation.

class deckhand.engine.document_validation.**DataSchemaValidator**(*data_schemas*)
Bases: *deckhand.engine.document_validation.GenericValidator*

Validator for validating `DataSchema` documents.

validate(*document*, *pre_validate=True*)
Validate `document` against built-in `schema`-specific schemas.

Does not apply to abstract documents.

Parameters

- **document** (`DocumentDict`) – Document to validate.
- **pre_validate** (*bool*) – Whether to pre-validate documents using built-in schema validation. Skips over externally registered `DataSchema` documents to avoid false positives. Default is True.

Raises **RuntimeError** – If the Deckhand schema itself is invalid.

Returns Tuple of (error message, parent path for failing property) following schema validation failure.

Return type Generator[Tuple[str, str]]

class deckhand.engine.document_validation.**DocumentValidation**(*documents*, *existing_data_schemas=None*, *pre_validate=True*)
Bases: `object`

validate_all()
Validate that all documents are correctly formatted.

All concrete documents in the revision must successfully pass their JSON schema validations. The result of the validation is stored under the "deckhand-document-schema-validation" validation namespace for a document revision.

All abstract documents must themselves be sanity-checked.

Validation is broken up into 2 "main" stages:

1) Validate that each document contains the basic bulding blocks needed: i.e. `schema` and `metadata` using a "base" schema. Failing this validation is deemed a critical failure, resulting in an exception.
2) Execute `DataSchema` validations if applicable. Includes all built-in `DataSchema` documents by default.

Returns A list of validations (one for each document validated).

Return type List[dict]

Raises

- ***errors.InvalidDocumentFormat*** – If the document failed schema validation and the failure is deemed critical.
- **RuntimeError** – If a Deckhand schema itself is invalid.

class deckhand.engine.document_validation.**DuplicateDocumentValidator**
Bases: *deckhand.engine.document_validation.BaseValidator*

Validator used for guarding against duplicate documents.

validate(*document*, ***kwargs*)
Validates that duplicate document doesn't exist.

class deckhand.engine.document_validation.**GenericValidator**
Bases: *deckhand.engine.document_validation.BaseValidator*

Validator used for validating all documents, regardless whether concrete or abstract, or what version its schema is.

base_schema

validate(*document*, ***kwargs*)
Validate `document` against basic schema validation.

Sanity-checks each document for mandatory keys like "metadata" and "schema".

Applies even to abstract documents, as they must be consumed by concrete documents, so basic formatting is mandatory.

Failure to pass this check results in an error.

Parameters **document** (*dict*) – Document to validate.

Raises

- **RuntimeError** – If the Deckhand schema itself is invalid.
- ***errors.InvalidDocumentFormat*** – If the document failed schema validation.

Returns None

validate_metadata(*metadata*)
Validate `metadata` against the given schema.

The `metadata` section of a Deckhand document describes a schema defining just the `metadata` section. Use that declaration to choose a schema for validating `metadata`.

Parameters **metadata** (*dict*) – Document metadata section to validate

Returns list of validation errors or empty list for success

deckhand.engine.layering module

class deckhand.engine.layering.**DocumentLayering**(*documents*, *validate=True*, *fail_on_missing_sub_src=True*, *encryption_sources=None*, *cleartext_secrets=False*)
Bases: `object`

Class responsible for handling document layering.

Layering is controlled in two places:

1. The `LayeringPolicy` control document, which defines the valid layers and their order of precedence.
2. In the `metadata.layeringDefinition` section of normal (`metadata.schema=metadata/Document/v1.0`) documents.

Note: Only documents with the same `schema` are allowed to be layered together into a fully rendered document.

---

documents

render()

Perform layering on the list of documents passed to __init__.

Each concrete document will undergo layering according to the actions defined by its metadata.layeringDefinition. Documents are layered with their parents. A parent document's schema must match that of the child, and its metadata.labels must much the child's metadata.layeringDefinition.parentSelector.

Returns The list of concrete rendered documents.

Return type List[dict]

Raises

- ***UnsupportedActionMethod*** – If the layering action isn't found among self.SUPPORTED_METHODS.
- ***MissingDocumentKey*** – If a layering action path isn't found in both the parent and child documents being layered together.

secrets_substitution

deckhand.engine.render module

deckhand.engine.render.**render**(*revision_id*, *documents*, *encryption_sources=None*, *cleartext_secrets=False*)

Render revision documents for revision_id using raw documents.

Parameters

- **revision_id** (*int*) – Key used for caching rendered documents by.
- **documents** (*List[dict]*) – List of raw documents corresponding to revision_id to render.
- **encryption_sources** (*dict*) – A dictionary that maps the reference contained in the destination document's data section to the actual unecrypted data. If encrypting data with Barbican, the reference will be a Barbican secret reference.
- **cleartext_secrets** (*bool*) – Whether to show unencrypted data as cleartext.

Returns Rendered documents for revision_id.

Return type List[dict]

deckhand.engine.render.**validate_render**(*revision_id*, *rendered_documents*, *validator*)

Validate rendered documents using validator.

Parameters

- **revision_id** (*int*) – Key used for caching rendered documents by.
- **documents** (*List[dict]*) – List of rendered documents corresponding to revision_id.
- **validator** (deckhand.engine.document_validation.DocumentValidation) – Validation object used for validating rendered_documents.

Raises InvalidDocumentFormat if validation fails.

deckhand.engine.revision_diff module

deckhand.engine.revision_diff.**revision_diff**(*revision_id*, *comparison_revision_id*, *deepdiff=False*)

Generate the diff between two revisions.

Generate the diff between the two revisions: *revision_id* and *comparison_revision_id*. a. When deepdiff=False: A basic comparison of the revisions in terms of how the buckets involved have changed is generated. Only buckets with existing documents in either of the two revisions in question will be reported. b. When deepdiff=True: Along with basic comparision, It will generate deep diff between revisions' modified buckets.

Only in case of diff, The ordering of the two revision IDs is interchangeable, i.e. no matter the order, the same result is generated.

The differences include:

- "created": A bucket has been created between the revisions.
- "deleted": A bucket has been deleted between the revisions.
- **"modified": A bucket has been modified between the revisions.** When deepdiff is enabled, It also includes deep difference between the revisions.
- "unmodified": A bucket remains unmodified between the revisions.

Parameters

- **revision_id** – ID of the first revision.
- **comparison_revision_id** – ID of the second revision.
- **deepdiff** – Whether deepdiff needed or not.

Returns A dictionary, keyed with the bucket IDs, containing any of the differences enumerated above.

Examples Diff:

```
# GET /api/v1.0/revisions/6/diff/3
bucket_a: created
bucket_b: deleted
bucket_c: modified
bucket_d: unmodified

# GET /api/v1.0/revisions/0/diff/6
bucket_a: created
bucket_c: created
bucket_d: created

# GET /api/v1.0/revisions/6/diff/6
bucket_a: unmodified
bucket_c: unmodified
bucket_d: unmodified

# GET /api/v1.0/revisions/0/diff/0
{}
```

Examples DeepDiff:

```
# GET /api/v1.0/revisions/3/deepdiff/4
bucket_a: modified
bucket_a diff:
  document_changed:
    count: 1
    details:
      ('example/Kind/v1', 'doc-b'):
        data_changed:
          values_changed:
            root['foo']: {new_value: 3, old_value: 2}
        metadata_changed: {}

# GET /api/v1.0/revisions/2/deepdiff/3
bucket_a: modified
bucket_a diff:
  document_added:
    count: 1
    details:
    - [example/Kind/v1, doc-c]

# GET /api/v1.0/revisions/0/deepdiff/0
{}

# GET /api/v1.0/revisions/0/deepdiff/3
bucket_a: created
```

deckhand.engine.secrets_manager module

class deckhand.engine.secrets_manager.**SecretsManager**

Bases: object

Internal API resource for interacting with Barbican.

Currently only supports Barbican.

barbican_driver = <deckhand.barbican.driver.BarbicanDriver object>

classmethod create(*secret_doc*)

Securely store secrets contained in secret_doc.

Documents with metadata.storagePolicy == "clearText" have their secrets stored directly in Deckhand.

Documents with metadata.storagePolicy == "encrypted" are stored in Barbican directly. Deckhand in turn stores the reference returned by Barbican in its own DB.

Parameters **secret_doc** – A Deckhand document with a schema that belongs to types.DOCUMENT_SECRET_TYPES.

Returns Unecrypted data section from secret_doc if the document's storagePolicy is "cleartext" or a Barbican secret reference if the storagePolicy is "encrypted'.

classmethod delete(*document*)

Delete a secret from Barbican.

Parameters **document** (*dict*) – Document with secret_ref in data section with format: "https://{barbican_host}/v1/secrets/{secret_uuid}"

Returns None

classmethod get(*secret_ref*, *src_doc*)
Retrieve a secret payload from Barbican.

Extracts {secret_uuid} from a secret reference and queries Barbican's Secrets API with it.

Parameters **secret_ref** (*str*) – A string formatted like: "https://{barbican_host}/v1/secrets/{secret_uuid}"

Returns Secret payload from Barbican.

static requires_encryption(*document*)

class deckhand.engine.secrets_manager.**SecretsSubstitution**(*substitution_sources=None*, *fail_on_missing_sub_src=True*, *encryption_sources=None*, *cleartext_secrets=False*)

Bases: object

Class for document substitution logic for YAML files.

get_unencrypted_data(*secret_ref*, *src_doc*, *dest_doc*)

static sanitize_potential_secrets(*error*, *document*)
Sanitize all secret data that may have been substituted into the document or contained in the document itself (if the document has metadata.storagePolicy == 'encrypted'). Uses references in document.substitutions to determine which values to sanitize. Only meaningful to call this on post-rendered documents.

Parameters

- **error** – Error message produced by jsonschema.
- **document** (DocumentDict) – Document to sanitize.

substitute_all(*documents*)
Substitute all documents that have a *metadata.substitutions* field.

Concrete (non-abstract) documents can be used as a source of substitution into other documents. This substitution is layer-independent, a document in the region layer could insert data from a document in the site layer.

Parameters **documents** (*dict or List[dict]*) – List of documents that are candidates for substitution.

Returns List of fully substituted documents.

Return type Generator[DocumentDict]

Raises

- ***SubstitutionSourceNotFound*** – If a substitution source document is referenced by another document but wasn't found.
- ***UnknownSubstitutionError*** – If an unknown error occurred during substitution.

update_substitution_sources(*meta*, *data*)
Update substitution sources with rendered data so that future layering and substitution sources reference the latest rendered data rather than stale data.

Parameters

- **meta** (*tuple*) – Tuple of (schema, layer, name).

- **data** (*dict*) – Dictionary of just-rendered document data that belongs to the document uniquely identified by `meta`.

Returns None

deckhand.engine.utils module

`deckhand.engine.utils.`**deep_delete**(*target*, *value*, *parent*)

Recursively search for then delete `target` from `parent`.

Parameters

- **target** – Target value to remove.
- **value** – Current value in a list or dict to compare against `target` and removed from `parent` given match.
- **parent** (*list or dict*) – Tracks the parent data structure from which `value` is removed.

Returns Whether `target` was found.

Return type bool

`deckhand.engine.utils.`**deep_merge**(*dct*, *merge_dct*)

Recursive dict merge. Inspired by :meth:`dict.update()`, instead of updating only top-level keys, deep_merge recurses down into dicts nested to an arbitrary depth, updating keys. The `merge_dct` is merged into `dct`, except for merge conflicts, which are resolved by prioritizing the `dct` value.

Borrowed from: https://gist.github.com/angstwad/bf22d1822c38a92ec0a9#file-deep_merge-py # noqa

Parameters

- **dct** – dict onto which the merge is executed
- **merge_dct** – dct merged into dct

Returns None

`deckhand.engine.utils.`**deep_scrub**(*value*, *parent*)

Scrubs all primitives in document data recursively. Useful for scrubbing any and all secret data that may have been substituted into the document data section before logging it out safely following an error.

`deckhand.engine.utils.`**exclude_deleted_documents**(*documents*)

Excludes all documents that have been deleted including all documents earlier in the revision history with the same `metadata.name` and `schema` from `documents`.

`deckhand.engine.utils.`**filter_revision_documents**(*documents*, *unique_only*, ***filters*)

Return the list of documents that match filters.

Parameters

- **documents** – List of documents to apply `filters` to.
- **unique_only** – Return only unique documents if `True`.
- **filters** – Dictionary attributes (including nested) used to filter out revision documents.

Returns List of documents that match specified filters.

`deckhand.engine.utils.`**meta**(*document*)

Module contents

deckhand.engine.**render**(*revision_id*, *documents*, *encryption_sources=None*, *cleartext_secrets=False*)

Render revision documents for `revision_id` using raw `documents`.

Parameters

- **revision_id** (*int*) – Key used for caching rendered documents by.
- **documents** (*List[dict]*) – List of raw documents corresponding to `revision_id` to render.
- **encryption_sources** (*dict*) – A dictionary that maps the reference contained in the destination document's data section to the actual unecrypted data. If encrypting data with Barbican, the reference will be a Barbican secret reference.
- **cleartext_secrets** (*bool*) – Whether to show unencrypted data as cleartext.

Returns Rendered documents for `revision_id`.

Return type List[dict]

deckhand.engine.**validate_render**(*revision_id*, *rendered_documents*, *validator*)

Validate rendered documents using `validator`.

Parameters

- **revision_id** (*int*) – Key used for caching rendered documents by.
- **documents** (*List[dict]*) – List of rendered documents corresponding to `revision_id`.
- **validator** (deckhand.engine.document_validation.DocumentValidation) – Validation object used for validating `rendered_documents`.

Raises InvalidDocumentFormat if validation fails.

deckhand.policies package

Submodules

deckhand.policies.base module

deckhand.policies.base.**list_rules**()

deckhand.policies.document module

deckhand.policies.document.**list_rules**()

deckhand.policies.revision module

deckhand.policies.revision.**list_rules**()

deckhand.policies.revision_tag module

deckhand.policies.revision_tag.**list_rules**()

deckhand.policies.validation module

deckhand.policies.validation.**list_rules**()

Module contents

deckhand.policies.**list_rules**()

Submodules

deckhand.context module

class deckhand.context.**RequestContext**(*project=None*, *context_marker='-'*, *end_user='-'*, ***kwargs*)

Bases: oslo_context.context.RequestContext

User security context object

Stores information about the security context under which the user accesses the system, as well as additional request information.

classmethod from_dict(*values*)
Construct a context object from a provided dictionary.

to_dict()
Return a dictionary of context attributes.

deckhand.context.**get_context**()
A helper method to get a blank context (useful for tests).

deckhand.errors module

exception deckhand.errors.**BarbicanClientException**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

A client-side 4xx error occurred with Barbican.

Troubleshoot:

- Ensure that Deckhand can authenticate against Keystone.
- Ensure that Deckhand's Barbican configuration options are correct.
- Ensure that Deckhand and Barbican are contained in the Keystone service catalog.

code = 400

msg_fmt = 'Barbican raised a client error. Details: %(details)s'

exception deckhand.errors.**BarbicanServerException**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

A server-side 5xx error occurred with Barbican.

code = 500

msg_fmt = 'Barbican raised a server error. Details: %(details)s'

exception deckhand.errors.**DeckhandException**(*message=None*, *code=500*, ***kwargs*)

Bases: Exception

Base Deckhand Exception To correctly use this class, inherit from it and define a 'msg_fmt' property. That msg_fmt will get printf'd with the keyword arguments provided to the constructor.

format_message()

msg_fmt = 'An unknown exception occurred'

exception deckhand.errors.**DeepDiffException**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

An Exception occurred while deep diffing

code = 500

msg_fmt = 'An Exception occurred while deep diffing. Details: %(details)s'

exception deckhand.errors.**DocumentNotFound**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

The requested document could not be found.

Troubleshoot:

code = 404

msg_fmt = 'The requested document using filters: %(filters)s was not found'

exception deckhand.errors.**DuplicateDocumentExists**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

A document attempted to be put into a bucket where another document with the same schema and metadata.name already exist.

Troubleshoot:

code = 409

msg_fmt = 'Document [%(schema)s, %(layer)s] %(name)s already exists in bucket: %(buck

exception deckhand.errors.**EncryptionSourceNotFound**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

Required encryption source reference was not found.

Troubleshoot:

- Ensure that the secret reference exists among the encryption sources.

code = 400

msg_fmt = 'Required encryption source reference could not be resolved into a secret be

exception deckhand.errors.**IndeterminateDocumentParent**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

More than one parent document was found for a document.

Troubleshoot:

code = 400

msg_fmt = 'Too many parent documents found for document [%(schema)s, %(layer)s] %(name

exception deckhand.errors.**InvalidDocumentFormat**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

Schema validations failed for the provided document(s).

Troubleshoot:

code = 400

msg_fmt = 'The provided documents failed schema validation'

exception deckhand.errors.**InvalidDocumentLayer**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

The document layer is invalid.

Troubleshoot:

- Check that the document layer is contained in the layerOrder in the registered LayeringPolicy in the system.

code = 400

msg_fmt = "Invalid layer '%(document_layer)s' for document [%(document_schema)s] %(doc

exception deckhand.errors.**InvalidDocumentParent**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

The document parent is invalid.

Troubleshoot:

- Check that the document *schema* and parent *schema* match.
- Check that the document layer is lower-order than the parent layer.

code = 400

msg_fmt = 'The document parent [%(parent_schema)s] %(parent_name)s is invalid for docu

exception deckhand.errors.**InvalidDocumentReplacement**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

The document replacement is invalid.

Troubleshoot:

- Check that the replacement document has the same `schema` and `metadata.name` as the document it replaces.
- Check that the document with `replacement: true` has a parent.
- Check that the document replacement isn't being replaced by another document. Only one level of replacement is permitted.

code = 400

msg_fmt = 'Replacement document [%(schema)s, %(layer)s] %(name)s is invalid. Reason:

exception deckhand.errors.**InvalidInputException**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

An Invalid Input provided due to which unable to process request.

code = 400

msg_fmt = 'Failed to process request due to invalid input: %(input_var)s'

exception deckhand.errors.**LayeringPolicyNotFound**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

Required LayeringPolicy was not found for layering.

Troubleshoot:

code = 409

msg_fmt = 'Required LayeringPolicy was not found for layering'

exception deckhand.errors.**MissingDocumentKey**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

Either the parent or child document data is missing the action path used for layering.

Troubleshoot:

- Check that the action path exists in the data section for both child and parent documents being layered together.
- Note that previous delete layering actions can affect future layering actions by removing a path needed by a future layering action.
- Note that substitutions that substitute in lists or objects into the rendered data for a document can also complicate debugging this issue.

code = 400

msg_fmt = 'Missing action path in %(action)s needed for layering from either the data

exception deckhand.errors.**MissingDocumentPattern**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

'Pattern' is not None and data[jsonpath] doesn't exist.

Troubleshoot:

- Check that the destination document's data section contains the pattern specified under *substitutions.dest.pattern* in its data section at *substitutions.dest.path*.

code = 400

msg_fmt = "The destination document's `data` section is missing the pattern %(pattern)

exception deckhand.errors.**PolicyNotAuthorized**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

The policy action is not found in the list of registered rules.

Troubleshoot:

code = 403

msg_fmt = "Policy doesn't allow %(action)s to be performed"

exception deckhand.errors.**RevisionNotFound**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

The revision cannot be found or doesn't exist.

Troubleshoot:

code = 404

msg_fmt = 'The requested revision=%(revision_id)s was not found'

exception deckhand.errors.**RevisionTagBadFormat**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

The tag data is neither None nor dictionary.

Troubleshoot:

code = 400

msg_fmt = 'The requested tag data %(data)s must either be null or dictionary'

exception deckhand.errors.**RevisionTagNotFound**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

The tag for the revision id was not found.

Troubleshoot:

code = 404

msg_fmt = "The requested tag '%(tag)s' for revision %(revision)s was not found"

exception deckhand.errors.**SingletonDocumentConflict**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

A singleton document already exist within the system.

Troubleshoot:

code = 409

msg_fmt = 'A singleton document [%(schema)s, %(layer)s] %(name)s already exists in the

exception deckhand.errors.**SubstitutionDependencyCycle**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

An illegal substitution depdencency cycle was detected.

Troubleshoot:

- Check that there is no two-way substitution dependency between documents.

code = 400

msg_fmt = 'Cannot determine substitution order as a dependency cycle exists for the fo

exception deckhand.errors.**SubstitutionSourceDataNotFound**(*message=None*, *code=500*, ***kwargs*)
Bases: *deckhand.errors.DeckhandException*

Required substitution source secret was not found in the substitution source document at the path `metadata.substitutions.[*].src.path` in the destination document.

Troubleshoot:

- Ensure that the missing source secret exists at the `src.path` specified under the given substitution in the destination document and that the `src.path` itself exists in the source document.

code = 400

msg_fmt = 'Required substitution source secret was not found at path %(src_path)s in s

exception deckhand.errors.**SubstitutionSourceNotFound**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

Required substitution source document was not found.

Troubleshoot:

- Ensure that the missing source document being referenced exists in the system or was passed to the layering module.

code = 409

msg_fmt = 'Required substitution source document [%(src_schema)s] %(src_name)s was not

exception deckhand.errors.**UnknownSubstitutionError**(**args*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

An unknown error occurred during substitution.

Troubleshoot:

code = 500

exception deckhand.errors.**UnsupportedActionMethod**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

The action is not in the list of supported methods.

Troubleshoot:

code = 400

msg_fmt = 'Method in %(actions)s is invalid for document %(document)s'

exception deckhand.errors.**ValidationNotFound**(*message=None*, *code=500*, ***kwargs*)

Bases: *deckhand.errors.DeckhandException*

The requested validation was not found.

Troubleshoot:

code = 404

msg_fmt = 'The requested validation entry %(entry_id)s was not found for validation na

deckhand.errors.**default_exception_handler**(*ex*, *req*, *resp*, *params*)

Catch-all exception handler for standardized output.

If this is a standard falcon HTTPError, rethrow it for handling by `default_exception_serializer` below.

deckhand.errors.**default_exception_serializer**(*req*, *resp*, *exception*)

Serializes instances of `falcon.HTTPError` into YAML format and formats the error body so it adheres to the Airship error formatting standard.

deckhand.errors.**format_error_resp**(*req*, *resp*, *status_code='500 Internal Server Error'*, *message=''*, *reason=None*, *error_type=None*, *error_list=None*, *info_list=None*)

Generate a error message body and throw a Falcon exception to trigger an HTTP status.

Parameters

- **req** – `falcon` request object.
- **resp** – `falcon` response object to update.
- **status_code** – `falcon` status_code constant.
- **message** – Optional error message to include in the body. This should be the summary level of the error message, encompassing an overall result. If no other messages are passed in the error_list, this message will be repeated in a generated message for the output message_list.
- **reason** – Optional reason code to include in the body
- **error_type** – If specified, the error type will be used; otherwise, this will be set to 'Unspecified Exception'.
- **error_list** – optional list of error dictionaries. Minimally, the dictionary will contain the 'message' field, but should also contain 'error': `True`.
- **info_list** – optional list of info message dictionaries. Minimally, the dictionary needs to contain a 'message' field, but should also have a 'error': `False` field.

deckhand.errors.**get_version_from_request**(*req*)

Attempt to extract the API version string.

deckhand.factories module

class deckhand.factories.**DataSchemaFactory**

Bases: *deckhand.factories.DeckhandFactory*

Class for auto-generating `DataSchema` templates for testing.

DATA_SCHEMA_TEMPLATE = {'data': {'$schema': ''}, 'metadata': {'labels': {}, 'layer

gen_test(*metadata_name*, *data*, ***metadata_labels*)

Generate an object with randomized values for a test.

class deckhand.factories.**DeckhandFactory**

Bases: `object`

gen_test(**args*, ***kwargs*)

Generate an object with randomized values for a test.

class deckhand.factories.**DocumentFactory**(*num_layers*, *docs_per_layer*)

Bases: *deckhand.factories.DeckhandFactory*

Class for auto-generating document templates for testing.

DOCUMENT_TEMPLATE = {'data': {}, 'metadata': {'labels': {'': ''}, 'layeringDefinit

LAYERING_POLICY_TEMPLATE = {'data': {'layerOrder': []}, 'metadata': {'layeringDefin

gen_test(*mapping*, *site_abstract=True*, *region_abstract=True*, *global_abstract=True*, *site_parent_selectors=None*)

Generate the document template.

Generate the document template based on the arguments passed to the constructor and to this function.

Parameters

- **mapping** (*dict*) – A list of dictionaries that specify the "data" and "actions" parameters for each document. A valid mapping is:

```
mapping = {
    "_GLOBAL_DATA_1_": {"data": {"a": {"x": 1, "y": 2}}},
    "_SITE_DATA_1_": {"data": {"a": {"x": 7, "z": 3}, "b": 4}},
    "_SITE_ACTIONS_1_": {
        "actions": [{"method": "merge", "path": path}]}
}
```

  Each key must be of the form "_{LAYER_NAME}_{KEY_NAME}_{N}_" where:

  - **{LAYER_NAME} is the name of the layer ("global", "region",** "site")
  - {KEY_NAME} is either "DATA" or "ACTIONS"
  - **{N} is the occurrence of the document based on the** values in `docs_per_layer`. If `docs_per_layer` is (1, 2) then _GLOBAL_DATA_1_, _SITE_DATA_1_, _SITE_DATA_2_, _SITE_ACTIONS_1_ and _SITE_ACTIONS_2_ must be provided. _GLOBAL_ACTIONS_{N}_ is ignored.

- **site_abstract** (*boolean*) – Whether site layers are abstract/concrete.
- **region_abstract** (*boolean*) – Whether region layers are abstract/concrete.
- **global_abstract** (*boolean*) – Whether global layers are abstract/concrete.
- **site_parent_selectors** (*list*) – Override the default parent selector for each site. Assuming that `docs_per_layer` is (2, 2), for example, a valid value is:

```
[{'global': 'global1'}, {'global': 'global2'}]
```

  If not specified, each site will default to the first parent.

Returns Rendered template of the form specified above.

class deckhand.factories.**DocumentSecretFactory**

Bases: *deckhand.factories.DeckhandFactory*

Class for auto-generating document secrets templates for testing.

Returns formats that adhere to the following supported schemas:

- deckhand/Certificate/v1
- deckhand/CertificateKey/v1
- deckhand/Passphrase/v1

DOCUMENT_SECRET_TEMPLATE = {'data': {}, 'metadata': {'layeringDefinition': {'abstra

gen_test(*schema*, *storage_policy*, *data=None*, *name=None*)

Generate an object with randomized values for a test.

class deckhand.factories.**RenderedDocumentFactory**(*bucket*, *revision*)

Bases: *deckhand.factories.DeckhandFactory*

Class for auto-generating Rendered document for testing.

RENDERED_DOCUMENT_TEMPLATE = {'data': {}, 'data_hash': '', 'metadata': {'layeringD

gen_test(*schema*, *name*, *storagePolicy*, *data*, *doc_no=1*)

Generate Test Rendered Document.

deckhand.policy module

deckhand.policy.**authorize**(*action*)

Verifies whether a policy action can be performed given the credentials found in the falcon request context.

Parameters **action** – The policy action to enforce.

Returns `True` if policy enforcement succeeded, else `False`.

Raises falcon.HTTPForbidden if policy enforcement failed or if the policy action isn't registered under `deckhand.policies`.

deckhand.policy.**conditional_authorize**(*action*, *context*, *do_raise=True*)

Conditionally authorize a policy action.

Parameters

- **action** – The policy action to enforce.
- **context** – The falcon request context object.
- **do_raise** – Whether to raise the exception if policy enforcement fails. `True` by default.

Raises falcon.HTTPForbidden if policy enforcement failed or if the policy action isn't registered under `deckhand.policies`.

Example:

```
# If any requested documents' metadata.storagePolicy == 'cleartext'.
if cleartext_documents:
    policy.conditional_authorize('deckhand:create_cleartext_documents',
                                 req.context)
```

deckhand.policy.**init**(*policy_file=None*, *rules=None*, *default_rule=None*, *use_conf=True*)

Init an Enforcer class.

Parameters

- **policy_file** – Custom policy file to use, if none is specified, `CONF.policy_file` will be used.
- **rules** – Default dictionary / Rules to use. It will be considered just in the first instantiation.
- **default_rule** – Default rule to use; `CONF.default_rule` will be used if none is specified.
- **use_conf** – Whether to load rules from config file.

deckhand.policy.**register_rules**(*enforcer*)

deckhand.policy.**reset**()

deckhand.service module

deckhand.service.**configure_app**(*app*, *version=''*)

deckhand.service.**deckhand_app_factory**(*global_config*, ***local_config*)

deckhand.types module

Module contents

4.1.4 Indices and tables

- genindex
- modindex
- search

CHAPTER 5

Release Notes

5.1 Deckhand Release Notes

5.1.1 Current Series Release Notes

0.0.0

New Features

- Development mode has been added to Deckhand, allowing for the possibility of running Deckhand without Keystone. A new paste file has been added to `etc/deckhand` called `noauth-paste.ini` which excludes Keystone authentication. To run Deckhand in development mode, set development_mode as True in deckhand.conf. Note that Deckhand will expect to find `noauth-paste.ini` on the host with development_mode set as True in etc/deckhand/deckhand.conf.sample.
- Adds a new endpoint to the Deckhand Validations API, GET /api/v1.0/{revision_id}/validations/detail, which allows for the possibility of listing all validations for a revision with details. The response body includes all details returned by retrieving validation details for a specific validation entry.
- The `oslo.policy` framework has been integrated into Deckhand. All currently supported endpoints are covered by RBAC enforcement. All default policy rules are admin-only by default. The defaults can be overridden via a custom `policy.yaml` file.
- Deckhand now supports the following filter arguments for filtering revision documents:
  - schema
  - metadata.name
  - metadata.label
  - metadata.layeringDefinition.abstract
  - metadata.layeringDefinition.layer
  - status.bucket

Deckhand now supports the following filter arguments for filtering revisions:

  - tag

- Deckhand now supports secret substitution for documents. The endpoint `GET revisions/{revision_id}/rendered-documents` has been added to Deckhand, which allows the possibility of listing fully substituted documents. Only documents with `metadata.substitutions` field undergo secret substitution dynamically.
- The Validations API has been introduced to Deckhand, allowing users to register new validation results in Deckhand, as well as query the API for validation results for a revision. The validation results include a list of errors that occurred during document validation.

  The following endpoints have been implemented:

  - /api/v1.0/revisions/{revision_id}/validations
  - /api/v1.0/revisions/{revision_id}/validations/{validation_name}
  - /api/v1.0/revisions/{revision_id}/validations/{validation_name}/entries
  - /api/v1.0/revisions/{revision_id}/validations/{validation_name}/entries/{entry_id}

Bug Fixes

- Deckhand will no longer throw an `AttributeError` after a `yaml.scanner.ScannerError` is raised when attempting to parse a malformed YAML document. Deckhand should now correctly raise a "400 Bad Request" instead.
- Deckhand will allow only one document with the schema `LayeringPolicy` to exist in the system at a time. To update the existing layering policy, the layerign policy with the same name as the existing one should be passed in. To create a new layering policy, delete the existing one first.
- Removed indentation rules E127, E128, E129 and E131 from pep8 exclusion.

CHAPTER 6

Glossary

6.1 Glossary

6.1.1 A

Airship Airship is a collection of interoperable and loosely coupled open source tools, among which is Deckhand, that provide automated cloud provisioning and management in a declarative way.

Alembic Database migration software for Python and SQLAlchemy based databases.

6.1.2 B

barbican Code name of the *Key Manager service*.

bucket A bucket manages collections of documents together, providing write protections around them. Any bucket can read documents from any other bucket.

6.1.3 D

document A collection of metadata and data in YAML format. The data document format is modeled loosely after Kubernetes practices. The top level of each document is a dictionary with 3 keys: *schema*, *metadata*, and *data*.

6.1.4 K

Key Manager service (barbican) The project that produces a secret storage and generation system capable of providing key management for services wishing to enable encryption features.

6.1.5 M

migration (databse) A transformation of a databse from one version or structure to another. Migrations for Deckhand's database are performed using Alembic.

6.1.6 S

SQLAlchemy Databse toolkit for Python.

Python Module Index

d

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

Y