neonHIVE Notes

# Introduction

This document is a bucket for information that may make it into more formal documentation later.

# Deployment

Docker 1.12.0 and later allows swarm mode services to be deployed on both manager and worker nodes. This is reasonable for small clusters, but we recommend that for larger and sensitive production clusters that most workloads be constrained to deploy only to worker nodes. This limits the potential for cluster services to overload manager nodes and potentially disrupt the entire cluster.

# Networks, Load Balancers, and Rules

neonHIVE hosts reserve ports 5000-5499 for internal cluster purposes, including: for services like the neon-registry-cache, the logging Elasticsearch cluster, as well as the neon-proxy-public and neon-proxy-private services described below. The port definitions are defined in the NeonClusterPorts class.

neonHIVEs deploy two proxy services that can be used to route TCP and HTTP/S traffic to services and containers attached to the built-in neon-public and neon-private networks. These proxies are based on the [neoncluster/neon-proxy](https://hub.docker.com/r/neoncluster/neon-proxy/) image which relies on [HAProxy](http://www.haproxy.org).

The neon-proxy-public load balancer is intended primarily for routing inbound Internet traffic to services and containers on the neon-public network.

neon-proxy-private is intended for routing internal cluster traffic or perhaps internal traffic between datacenters to the neon-private network.

These load balancers are used to augment the built-in Docker service discovery and routing capabilities which doesn’t provide for routing external traffic from a router or load balancer to Docker services and also for providing load balancing for stateful services implemented as a group of individual (non-service) containers.

Load balancers are configured using the neon-cli by managing rules and TLS certificates via the load-balancer and cert commands. The built-in [neon-proxy-manager](https://hub.docker.com/r/neoncluster/neon-proxy-manager/) service watches for these changes and rebuilds the load balancer configurations as required.

Note that network, proxy, and rule names prefixed by neon-\* are reserved for neonHIVE use.

# Docker Secrets

As of version 1.13.0, Docker supports secrets for swarm mode services. Docker secrets are created by piping the secret (text or data) to the docker secret NAME command. This persists the secret in Docker using the NAME passed. The necessary secrets must be made available to cluster services as they are deployed.

Secret names prefixed by neon-\* are reserved for neonHIVE services. Here are the current neonHIVE secrets:

neon-proxy-manager-credentials Vault credentials for the neon-proxy-manager service.

neon-proxy-public-credentials Vault credentials for the neon-proxy-public service.

neon-proxy-private-credentials Vault credentials for the neon-proxy-private service.

# Vault

neonHIVE uses HashiCorp Vault for secret management. This is deployed to the cluster manager nodes and are configured for high-availability. The Vault unseal keys and root token are persisted anywhere in the cluster but instead, are encrypted and persisted on the operator’s workstation by the **neon-cli**.

## Default Vault Configuration

Vault is deployed initially with:

* **Auditing** to **syslog** is enabled using the **AUTH** facility and the **vault** tag.
* **Transit** backend is enabled to provide encryption as a service.
* A secret backend is mounted at **neon-secret** and is reserved for use by neonHIVE.
* The standard **/secret** backend is available for general use.
* The **AppRole** authentication backend is enabled.
* neonHIVE specific roles and policies are defined with names prefixed by **neon-\***.

## Naming Conventions

We recommend that you adhere to these naming conventions:

* The **neon-secret** secret backend is reserved for neonHIVE related services.
* Role, policy, transit or other Vault item names prefixed by **neon-\*** are reserved for use by neonHIVE related uses.
* To avoid conflicts, applications and services should prefix their Vault item names using a DNS host name you control (replacing dots with dashes). For example, the owner of the foo.com domain could:  
  + Store secrets at: **secret/foo-com/my-secrets**
  + Name an AppRole: **foo-com-myrole**
  + Name a policy: **foo-com-mypolicy**

## neonHIVE Secrets Hierarchy

neonHIVE secrets are persisted under the **/neon-secret** mount point using the follow structure.

neon-secret

global  
 dblogin: <credentials>

cert  
 mycert1: <cert + private key PEM encoded>  
 mycert2: <cert + private key PEM encoded>  
  
 hosting  
 options: <cluster hosting options>  
  
 registry/credentials:  
 registry-1.docker.io: username/password  
 registry.mycluster.com: username/password  
 ...

service  
 neon-foo  
 foo-secret1  
 foo-secret2  
 ...  
 neon-bar  
 bar-secret1  
 bar-secret2  
 ...  
 vpn  
 ca.zip.encrypted (zipped/encrypted certificate authority files)

Global secrets, like shared API keys or database credentials are saved under **neon-secret/global**.

Named TLS certificates are saved under **neon-secret/cert** with the value being the certificate concatenated with the private key, both PEM encoded.

Docker Registry related credentials will be stored beneath neon-secret/registry/credentials. This will be organized by target registry host name followed by the username and password used to log into each registry. Note that the credentials are persisted as UTF-8 and then encoded as base-64

Service specific secrets are saved under **neon-secret/service** using the service name. The hierarchy beneath this is defined by the service.

## neonHIVE Policies

neonHIVE policy names are prefixed by **neon-\***. Here’s the current list of policies initialized for a cluster:

**neon-reader** Can read/list any secret under **neon-secret/\***

**neon-reader** Can read/list any secret under **neon-secret/\***

**neon-cert-reader** Can read/list certificates.

**neon-cert-writer** Can read/write/list certificates.

**neon-hosting-reader** Can read/list cluster hosting information/credentials.

**neon-hosting-writer** Can read/write/list cluster hosting information/credentials.

**neon-service-reader** Can read/list any secret under **neon-secret/service/\***

**neon-service-writer** Can read/write/list secrets under **neon-secret/service/\***

**neon-global-reader** Can read/list any secret under **neon-secret/global/\***

**neon-global-writer** Can read/write/list secrets under **neon-secret/global/\***

TODO: Should we define policies for specific services or just set these explicitly in the AppRole when the service is deployed?

## neonHIVE AppRoles

neonHIVE role names are prefixed by **neon-\*** and typically match the name of the corresponding service. These roles are currently created when a cluster is provisioned:

**neon-proxy-public** Granted the **neon-cert-reader** policy to access the cluster certificates.

**neon-proxy-private** Granted the **neon-cert-reader** policy to access the cluster certificates.

# Consul

neonHIVE uses HashiCorp Consul as its primary key/value store. Consul servers are deployed to the cluster manager nodes and are configured for high-availability. Consul proxy agents are deployed to all worker nodes[[1]](#footnote-1) such that host processes and containers can always query the local proxy to access Consul.

## Layout

neonHIVE reserves the use of all keys prefixed by **neon\***.

We recommend that user-deployed applications and services prefix their keys using a DNS domain you control for example, the owner of the **foo.com** domain could prefix keys for the **bar** service like: **foo.com/bar/\***. Potential conflicts will be avoided by following this convention.

## Calling Consul

Consul can be accessed directly from each host using the CLI tool. The **CONSUL\_HTTP\_ADDR** environment variable and the local hosts file are configured such that no additional parameters are required. These settings can be propagated to containers and services that mount the host **/etc/neoncluster/env-host** script file and then execute it during startup so that the CLI can be used when installed into the container.

Other tools like **curl** may be used to query Consul. These tools should use the **CONSUL\_HTTP\_FULLADDR** environment variable because it includes the URL scheme (http:// or https://).

## neonHIVE Consul Hierarchy

neonHIVE reserves the use of all **neon\*** and **vault\*** key prefixes. Here’s structure beneath:

vault: - HashiCorp Vault data  
  
neon:  
 cluster:  
 allow-unit-testing - enables HiveFixture unit testing (bool)  
 create-date-utc - date/time the cluster was created (UTC)  
 definition-deflated – (json/compressed) current cluster definition  
 definition-hash - MD5 hash of the definition (base64)  
 disable-auto-unseal - disables Vault auto unsealing (bool)  
 log-retention-days - number of days to retain logs  
 neon-cli - minimum allowed client version  
 pets-definition - definitions for the cluster pets  
 uuid - UUID for the cluster  
 version - cluster version (actually the version of   
 neon-cli that created or last updated the  
 cluster  
  
 service:  
 neon-proxy-public  
 ...  
 neon-proxy-private  
 ...  
 neon-proxy-manager  
 ...

Non-sensitive cluster information that needs to be shared across multiple services will be persisted in keys under **cluster**. Keys used for global locks or semaphores will persisted under **lock**, and keys used to signal global events will be persisted under **event**.

Service specific keys will be located under **service/<service name>**.

# Logging

neonHIVE relies on TD-Agent (from [Fluentd](http://www.fluentd.org)), Elasticsearch, and Kibana for its logging infrastructure. We choose TD-Agent for log capture rather than Logstash because TD-Agent was much lighter weight then Logstash which required the Java runtime.

neonHIVE deploys several components to pull this all together:



The neon-log-host image is deployed as a local container on every cluster node (both managers and workers). It’s responsible for reading events from the systemd journal and receiving local syslog events as well as logs from local Docker containers configured to use the fluent log driver. The neon-log-host container filters out logs from itself, adds some datacenter, cluster, and node properties to each event and forwards the events on to the neon-log-collector service for further processing and storage

The neon-log-collector service is responsible for receiving raw events from the hosts, normalizing them into a standard format, and then persisting them to Elasticsearch.

## Host and Docker Container Statistics

neonHIVE uses Elastic [Metricbeat](https://www.elastic.co/products/beats/metricbeat) to capture Docker host node as well as Docker container statistics such as CPU, memory, disk I/O, etc. This is accomplished by deploying the [neoncluster/metricbeat](https://hub.docker.com/r/neoncluster/metricbeat/) image as a container on all cluster nodes.

## Log Record Formats

neonHIVEs persist four basic types of log events to the log Elasticsearch cluster: service, proxy, vault-audit, and Metricbeat. Service events capture typical status events emitted by services. These are persisted using the standard ELK stack Logstash format. Proxy events are emitted by HAProxy to describe network traffic handled by the cluster’s public, private, and Vault network proxies. Metricbeat events describe the status of a Docker host node or container at a point in time.

The following subsections describe the fields for the service and proxy events. You can examine the Elastic Metricbeat documentation for more information on those events.

### Service Event Fields

This section describes the standard neonHIVE fields for service events. The log pipeline (specifically the neon-log-collector service) is responsible for transforming raw received events into this format.

activity\_id Optional globally unique activity ID.

audit Optional field added for audit related events (see below).

cluster Identifies the source cluster within the datacenter.

container\_id Short container ID or empty if not a container.

datacenter Identifies the source datacenter.

environment Identifies the operational environment, one of: dev, test, stage, prod, or other.

index Optional long index of the generated event since the service was started. This is useful for displaying events in the actual order they were emitted by services that support this.

json Original raw string for event messages formatted as structured JSON.

level Event level/priority. See the description below for the possible values.

location Optional field that describes a related physical location. See the section below for more information.

message Optional event message text.

module Identifies the source component or application area.

node Docker host node name as specified in the cluster definition.

node\_dnsname Docker host node DNS name (if any).

node\_ip Docker host node’s IP address on the local cluster network.

node\_role Docker host node role, one of: manager or worker.

proxy Optional field added for proxy traffic events (see below).

service Names the service or container.

service\_host Identifies the how the service is hosted: system, docker,…

service\_type Identifies the underlying service type, e.g. elasticsearch, couchbase, mssql,… This may be the same as the service field in situations where the application code is single use.

tag The Fluentd event tag at the time when the event was persisted to Elasticsearch.

@timestamp Unix time (seconds) when the event was emitted or captured.

The possible level field values are inspired by the standard [syslog levels](https://en.wikipedia.org/wiki/Syslog#Severity_level):

emergency System is unusable (not emitted by services).

alert System is seriously degraded (not emitted by services)..

critical Service has failed (maps from fatal for Log4Net oriented application logging).

error Service has encountered an error.

warn Indicates that an error may occur if actions are not taken.

notice Something unusual has occurred but is not an error.

info Normal operational messages that require no action.

debug Developer/diagnostic information.

unknown Log level could not be determined.

### Location Details

Some events may be associated with a physical location. A common example are proxy traffic events where a reverse geographical lookup has been performed on the client IP address.

Note that geoname\_id fields refer to places in the [geonames.org](http://www.geonames.org/about.html) database.

The possible location fields:

latitude Latitude coordinate.

longitude Longitude coordinate.

metro\_code (USA only). [Nielson Designated Market Area ID](https://support.google.com/richmedia/answer/2745487?hl=en).

postal\_code Postal code.

time\_zone Timezone name.

continent Continent related fields.

code Two character continent code.

geoname\_id Name ID.

name Continent name.

country Country related fields.

geoname\_id Name ID.

iso\_code Three character ISO code.

name Country name.

city City related fields.

geoname\_id Name ID.

name City name.

subdivisions Array of regional subdivisions between country and city (e.g. State, County,…)

geoname\_id Name ID.

iso\_code ISO code.

name Subdivision name.

### Proxy Traffic Details

neonHIVEs deploy three proxy services: neon-proxy-vault, neon-proxy-public and neon-proxy-private. These services are based on HAProxy and handle TCP and HTTP load balancing for situations where it’s not possible to use the Docker ingress network. The logging pipeline recognizes log events from these services and adds the proxy field with traffic statistics to the event.

Common Fields for TCP and HTTP Traffic

Here are the proxy common subfields across TCP and HTTP traffic:

browser Optional field describing known browsers.

bot Indicates whether the browser is a known bot (true/false).

device Device name.

name Name of a known browser.

os\_name Operating system (platform) name.

os\_version Operating system (platform) version.

version Version of a known browsers.

bytes\_received Number of bytes received by the proxy from the client.

bytes\_sent Number of bytes sent by the proxy to the client.

client\_ip Client IP address.

conn\_proxy Total number of connections being handled by the proxy at the time the event was logged.

conn\_frontend Number of requests handled by the proxy frontend at the time the event was logged.

conn\_backend Number of proxy connections to internal HAProxy backend at the time the event was logged.

conn\_server Number of proxy connections to backend servers at the time the event was logged.

mode Identifies the type of proxy: tcp or http.

queue\_server Number of operations queued to be delivered to the current backend server at the time the event was logged.

queue\_backend Number of operations queued to the current proxy across all backend servers at the time the event was logged.

retries Number of times the operation was retried after an initial failure.

route Identifies the neonHIVE proxy route.

server Identifies the backend server. This can be configured in the route or will auto generated by the neon-proxy-manager.

server\_ip Backend server IP address.

server\_port Backend server port.

time\_queue Total seconds the operation waited in all queues.

time\_connect Seconds the operation waited for a connection to a backend server.

time\_session Seconds for the overall session.

termination Details how the session was terminated. See the HAProxy documentation [here](http://cbonte.github.io/haproxy-dconv/1.7/configuration.html#8.5) for more information.

tls\_version TLS version or the empty string if the connection is not secured.

tls\_cypher TLS cypher or the empty string if the connection is not secured.

Extended Fields for HTTP Traffic

Here are the extended proxy common subfields for and HTTP traffic:

http\_host HTTP Host header.

http\_method HTTP request method, like: GET, PUT, POST,…

http\_status HTTP response status code.

http\_time\_active Seconds from the request time (@timestamp) until the response was transmitted.

http\_time\_idle Seconds waiting idle for the first byte of the HTTP.

http\_time\_request Seconds to receive the full HTTP request from the first byte.

http\_time\_response Seconds the backend server took to process the request and return the response line and headers. This does not include the time to return all of the response contents.

http\_uri Relative request URI excluding query strings.

http\_uri\_query Request URI query string.

http\_user\_agent HTTP User-Agent header.

http\_version The HTTP version, like: HTTP/1.1

### Audit Details

TODO: DESCRIBE THESE

cluster Identifies the source cluster within the datacenter.

datacenter Identifies the source datacenter.

details Vault audit event details.

environment Identifies the operational environment, one of: dev, test, stage, prod, or other.

node Docker host node name as specified in the cluster definition.

node\_dnsname Docker host node DNS name (if any).

node\_ip Docker host node’s IP address on the local cluster network.

node\_role Docker host node role, one of: manager or worker.

service Names the service or container.

tag The Fluentd event tag at the time when the event was persisted to Elasticsearch.

@timestamp Unix time (seconds) when the event was emitted or captured.

## Custom Event Processing

By default, neonHIVE log pipeline attempts to extract some fields such as the timestamp, log level, module, and the remaining message from log events passing through the pipeline. This can work for many applications, but sometimes logs require custom processing.

To accomplish this, you’ll need to tag the application events with a log source identifier as the leading tag segment. The typical case will be to specify --log-driver=fluentd and then --log-opt tag=<value> as one of these strings when deploying a Docker service or container. This gives the neon-log-collector service enough information to customize event parsing for the specific application.

Here are the predefined identifiers supported by neonHIVE out of the box. You may specify custom tags and then extend the neon-log-collector image to support other applications.

neon-common Many neonFORGE applications emit a common log message format that include an optional timestamp, optional log-level, and optional sections in that order, followed by the message text. Each of the optional sections are surrounded by square brackets and are formatted like:  
  
[2017-01-27T19:04:11.000+00:00] [info] [module:main] [activity:<guid>] The message.

elasticsearch Elasticsearch cluster node.

# Activity Correlation

It is often useful to be able to correlate information about the actions performed across multiple systems and services to perform a high-level activity. neonHIVE handles this via globally unique activity IDs. By default, the neonHIVE HAProxy services will add an X-Activity-ID header to each proxied request, if this header isn’t already present. Standard services will be coded such that activity IDs from received requests will be forwarded on to downstream service calls and also included in the service logs.

Activity IDs are to be considered to be opaque strings. IDs generated by HAProxy for received requests without an activity ID will look like:  
  
C140001:C1B8\_AC140014:2327\_55F1BFF3\_0003

IDs generated by neonHIVEs related services will have .NET standard GUIDs like:  
  
2A45FAE8-9902-4F8B-BF58-0E4785D7519F

Other services may generate activity IDs with differing formats, so you should never depend on a specific format.

# Cluster Networks

neonHIVE assumes that the cluster servers are deployed on a network behind a firewall and/or router. The services will be assigned static IP addresses that cannot be changed after the cluster is deployed.

The neon tool configures and controls the servers by connecting directly to cluster servers using SSH/SCP and issuing commands. It also used HTTP/S to perform operations against various servers. This implies that the tool must have access to the cluster network. This can be achieved by:

1. Running the tool from a workstation on the cluster network.
2. Deploying a custom VPN solution.
3. Using the Built-in neonHIVE Management VPN.

Solution #1 works well for development and test environments, for very secure production environments where physical access is required or where a [jumpbox](https://en.wikipedia.org/wiki/Jump_server) is available for operators to log into remotely to administer the cluster. This will be problematic for other environments such as public clouds, where direct connection is impossible and jumping is inconvenient.

Solution #2 is certainly possible, effective, and secure but can be difficult and expensive to configure, especially for smaller environments. This can be accomplished using dedicated hardware and software as well as VPN services provided by the cloud environments.

Solution #3 is the easiest way to get started. This uses [OpenVPN](https://openvpn.net/) to establish a point-to-point VPN connection between the operator workstation and the cluster and the neon tool handles all of the provisioning!

## neonHIVE VPN

For cloud deployments, the neon tool will need to configure a VPN for cluster management purposes. This is not currently supported for private data center deployments, but will be in the future). This is based on [OpenVPN](https://openvpn.net/) and is intended to allow cluster operators to connect to the cluster network to perform operations there. This is not intended to serve as a general-purpose site-to-site VPN (although I suppose, it could be used that way in a pinch).

The management VPN can be enabled at cluster deployment time by setting the cluster configuration property:

vpn.enabled = true

The VPN servers are deployed to the cluster manager nodes. This implies that these nodes must have IP addresses that are reachable from the operator’s network. For cloud environments, the **neon** tool configures a load balancer with NAT rules to route VPN traffic through to the servers.

# Cluster DNS

neonHIVE provides built-in services such as Vault, Consul, Elasticsearch, and the Docker registry cache that are available for use by built-in and custom user Docker services and containers as well as native services running on the Docker host nodes. These services are typically accessed using host names using the custom \*.cluster domain, such as neon-consul.cluster, neon-vault.cluster, neon-log-esdata.cluster, etc.

Many of these host names need to resolve to the private IP address of the host node running the calling service or container. For services like Consul, this works because each node has a Consul instance running which a member of the Consul cluster. Other services have load balancers running on each node or will take advantage of the Docker ingress/mesh network (when that works). The basic idea here is that applications can simply send traffic to the host node its running on and rely on Consul, a load balancer, or Docker to forward the traffic to a functioning service somewhere in the cluster.

Older versions of neonHIVE configured static host mappings written to /etc/hosts during cluster setup. This works fine for DNS lookups made by native host node processes. The problem was how to get these mappings into Docker containers. To accomplish this, cluster setup created a static /etc/neoncluster/env-host script on the host and then we explicitly mounted this into containers and services and then had the container entry point execute the mounted script. This script set some environment variables and also wrote static host mappings to the container’s /etc/hosts file.

This is a little hokey though, since this means that containers will need to mount /etc/hosts to function, which is not in the spirit of Docker. A better approach would be to configure a dynamic DNS infrastructure that would handle these resolutions via standard DNS lookups.

Modern neonHIVE releases use the PowerDNS Recursor to accomplish the same thing more cleanly. Here’s how this works:



All manager, worker, and pet nodes install PowerDNS Recursor:

* Recursor is installed as a native host service on each node and acts as the immediate upstream server for both host and containers running on that node. Recursor is configured to listen on port 53 on all of the node network interfaces.
* Recursor is configured to authoritatively respond to requests for domains that match \*.cluster from a custom hosts file: /etc/powerdns/hosts.
* Recursor is configured to use PowerDNS Server installed as cluster managers as upstream DNS servers.
* The manager PowerDNS servers are configured use a remote backend to submit queries to the neon-dns service or to forward requests to upstream servers specified in the cluster configuration (this defaults to the Google name servers at 8.8.8.8 and 8.8.4.4).
* The DNS resolver on every host node will be configured to use the local PowerDNS Recursor as its upstream DNS server. Local Docker daemons will also be configured to use the local recursor by default to resolve container DNS queries.
* The Recursors running on each cluster node are configured to accept requests only from well-defined private Internet subnets.

This solution is pretty clean from an architectural perspective. Any DNS queries made by local host processes as well as by local containers will be received by the Recursor with \*.cluster questions being answered from the local zone files. This gives the local Server the chance to answer with the node’s private IP address, accomplishing our primary goal. Other requests will be forwarded to the PowerDNS servers running on the masters, which will perform a dynamic lookup or forward the query upstream.

## Dynamic DNS

The basic idea here is to deploy a couple services that dynamically update the PowerDNS Recursor’s local hosts file on the manager nodes with DNS entries read from Consul.

We’re going to implement this using two new services:

neon-dns This Docker service is deployed on all manager nodes that polls Consul for DNS answers and updates the local PowerDNS Recursor hosts file.

neon-dns-mon A single replica of this service is deployed. This must be deployed on a cluster manager so it can query the Docker Swarm REST API. This service performs health checks on DNS target endpoints as well as resolving cluster node groups into the set of currently valid DNS answers that will be persisted to Consul.

Both services access DNS state from Consul at:

neon/dns:  
 entries:  
 registry.mycluster.com  
 mysite1.com  
 mysite2.com  
 answers:  
 hosts.txt  
 hosts.md5

The keys under entries each specify a DnsEntry serialized as JSON. Each of these defines a dynamically resolvable DNS entry that specifies the DNS hostname, target endpoints and health checking options. Each entry is persisted to Consul using the *hostname* as the key (lowercase) for entries created by end users and **[neon]-***hostname* for system entries.

The neon-cli dns commands or neon-dns Ansible module is used to manage the DNS entry definitions in Consul.

The neon-dns-mon service reads the DNS entries from Consul, determines which of the target endpoints are healthy and then persists the answers as in /etc/hosts format to hosts.txt along with the MD5 hash of the hosts files as hosts.md5.

neon-dns simply polls Consul for changes to the DNS answers written by neon-dns-mon and updates the local hosts file when there are changes.

The PowerDNS Recursor running on each manager node simply attempts to resolve names from the local hosts file before forwarding requests on to the upstream DNS servers. This is radically less complex and fundamentally more reliable than my previous approach of using the authoritative server and a remote backend service or even trying to do something similar using Lua to intercept requests in the recursor.

Here's how this looks:



DNS host entries can be configured to reference specific endpoints by IP address, the cluster nodes in a host group or a fully qualified domain name which acts kind of like a CNAME record in normal DNS zone file.

Important: This CNAME like feature should be limited to reference DNS names that you control and are relatively stable. Referencing a host like [www.google.com](http://www.google.com) will cause the DNS to churn, because the Google DNS will likely return a single different IP address for every lookup and this change will cause a new hosts.txt to be uploaded the cluster Consul and loaded on each of the managers.

# Cluster Dashboards

neonHIVE provides commands to register dashboard URLs that can be launched using the neon-cli. Eventually, these dashboards may also be included in a global neonHIVE dashboard.

Dashboards are specified by the ClusterDashboard class. Currently, this class holds the dashboard URL. Dashboards are persisted to Consul by name and JSON specification like:

neon/dashboards:  
 kibana: { "Url": "http://healthy-manager.cluster:5003" }  
 ceph: { ... }  
 consul: { ... }

# Cluster CRON

neonHIVE includes the neon-cluster-cron service which can be configured to start one or more Docker services[[2]](#footnote-2) at scheduled times or periodic intervals. The neon-cluster-cron service runs as a single instance on the cluster managers (so it can access the Docker Swarm API) and works by polling and updating Consul keys beneath neon/service/neon-cluster-cron.

Here’s an example of how this lays out:

neon/service:  
 neon-cluster-cron:  
 poll-seconds: 15  
 disposition-poll-seconds: 5  
 disposition-wait-seconds: 120  
 history-count: 10  
  
 jobs:  
 backup:  
 schedule: Daily:01:00  
 command: docker service create ...  
 status: running  
 service-id: 9mnpnzenvg8p8tdbtq4wvbkcz  
 history:  
 9mnpnzenvg8p8tdbtq4wvbkcz: <job state>  
 7132nzenvg8p8tdbtq4wvbkcz: <job state>  
 ...  
  
 report:  
 schedule: Hourly  
 docker service create ...  
 status: waiting  
 service-id:   
 history:  
 ...

Service settings:

poll-seconds Specifies the interval at which neon-cluster-cron wakes up to decide when to launch new jobs as well as to manage running jobs.

disposition-poll-seconds Specifies the interval neon-cluster-cron will query Elasticsearch for the final disposition information for completed jobs.

disposition-wait-seconds Specifies the maximum number of seconds neon-cluster-cron will query Elasticsearch for the final disposition information for completed jobs.

history-count Specifies the maximum execution history records to retain in Consul for each job.

jobs Holds the job definitions and execution history keyed by job name.

Each cluster CRON job is defined by a record beneath jobs and is keyed by the job name. Job definitions include the following keys:

schedule Specifies when the job should be run. See the neonFORGE RecurringTimer class for more information.

command Specifies the command to be executed to start the job. Currently, only   
docker service create commands are supported. Note that the job name will be used when creating the service if the --name option is not specified.

status Indicates the current job status. The possible values are:  
  
waiting – job is waiting to be scheduled for execution  
running – job is currently running  
disabled – job has been disabled  
pending – job should be executed once immediately, ignoring the schedule

service-id Set to the associated Docker service ID when the job is running.

history Records describing the execution history of the job. Each record is keyed by the service-id of the service instance created for each job and the record is JSON text formatted as described below:

Each history record is formatted as JSON like:

{  
 "time": "2017-02-04T12:01:00.000",  
 "started-at": "2017-02-04T12:00:00.000",  
 "finished-at": "2017-02-04T12:01:00.000"  
 "disposition": "success",  
 "message": "success: Backup completed successfully!"  
}

where:

time Indicates the time (UTC) when the record was written

started-at Indicates the time (UTC) when the job was started

finished-at Indicates the time (UTC) when the job was finished

disposition Indicates whether the job is currently running or whether it succeeded or failed. The possible values are:  
  
running – job is currently running  
success – job finished successfully  
fail – job finished but indicated failure  
unknown – the job finished but its final disposition could not be determined

message Optional status message returned by the job.

## CRON Requirements and Limitations

At this point, a neon-cluster-cron job can only be executed as a Docker Swarm service running with a single replica. Global mode is not supported and it is also not currently possible to schedule jobs for pet nodes because they are not members of the Swarm. Future releases of neonHIVE may relax these constraints.

A cluster CRON job may be implemented using any Docker image. It’ll simply be scheduled to run as a Docker service. If possible, you should augment job images to write a special log entry to stdout or stderr indicating whether the job succeeded or failed and including an optional status message.

Successful jobs should output a log entry with a message like:

[INFO] NEON-JOB: Backup completed successfully.

Failed jobs should output a log entry with a message like:

[ERROR] NEON-JOB: Backup failed.

These log messages will eventually make their way into the Elasticsearch log cluster. Once  
neon-cluster-cron detects that a job service has terminated, it starts querying Elasticsearch for a log message prefixed by “NEON-JOB:”. Jobs that logged this as INFO will be considered to have been completed successfully, and jobs that logged ERROR will be considered to have failed. In either case, the message after the prefix will be extracted and be written to the message history property.

After a job service has completed, neon-cluster-cron will query Elasticsearch for the disposition log entry for up to disposition-wait-seconds. When a log record is found, the job history disposition will be set to success or fail based on the log level and the history message will be set to the extracted message. If no log record is found in time, the disposition will be set to unknown.

## neon-cli CRON commands

neonHIVE CRON jobs are managed using the following neon-cli commands:

### neon cron disable NAME

Disables execution of the named job.

### neon cron enable NAME

Enables execution of the named job.

### neon cron history [NAME]

Returns the execution history of all jobs or just the named job.

### neon cron ls|list

Lists the cron jobs.

### neon cron rm|remove NAME

Removes the named cron job.

### neon cron run NAME

Runs the named command immediately if it’s not already running.

### neon cron set NAME SCHEDULE -- docker service create …

Adds or updates a named job. SCHEDULE specifies when the job is to be executed and the command after the -- will be executed on one of the manager nodes to start the job. Only docker service create commands are supported at this time.

### neon cron status NAME

Returns the status of a named job.

1. This may change in the future. We may deploy an HAProxy instance to use the Docker ingress network instead. [↑](#footnote-ref-1)
2. In the future, I also expect that we’ll support launching containers on specific host nodes as well by connecting to the nodes via SSH and invoking the docker run command locally. [↑](#footnote-ref-2)