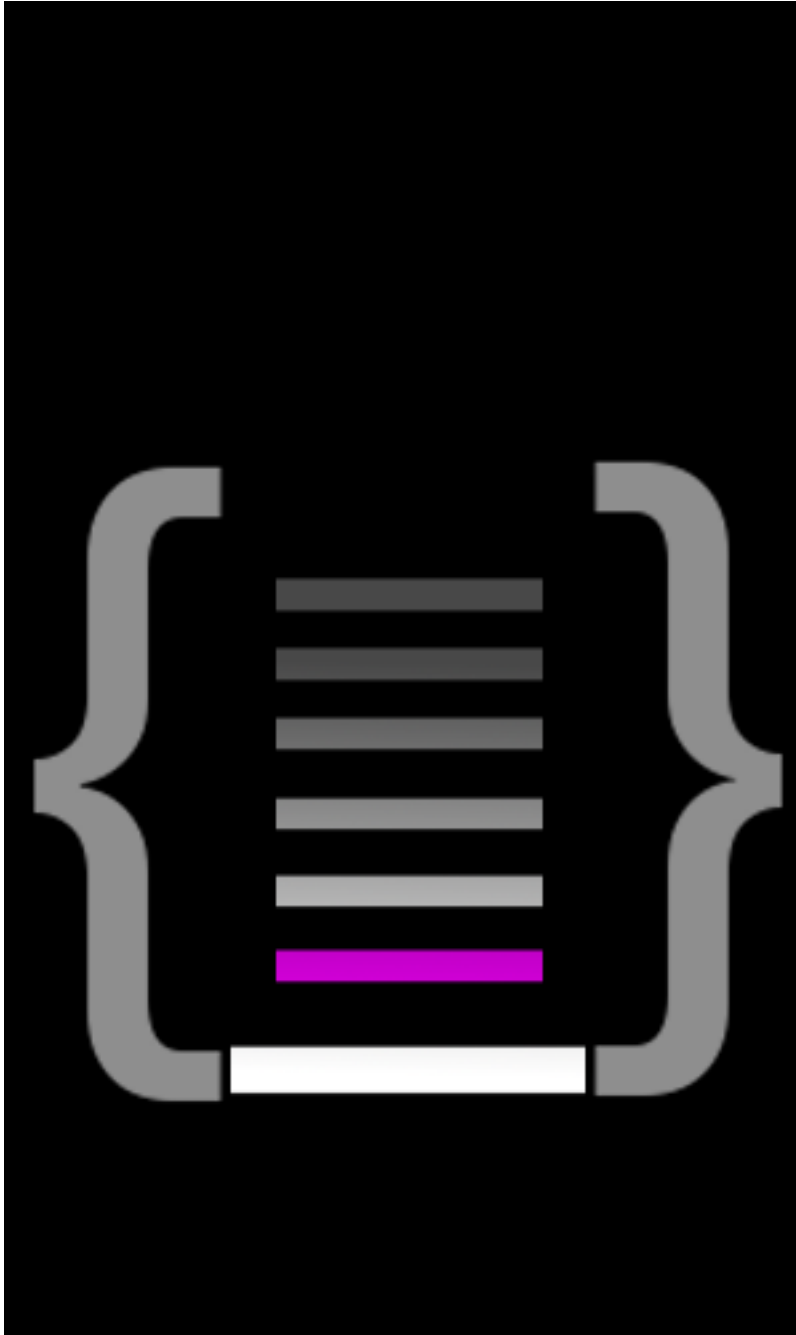

RackHD Documentation

Release 2.0

Dell EMC

April 26, 2018

1	Contents	3
1.1	RackHD Overview	3
1.2	System Architecture	7
1.3	Features	12
1.4	Contributing to RackHD	14
1.5	RackHD Development Guide	17
1.6	RackHD Users Guide	32
1.7	Tutorials	192

**VIDEO:** [Introduction to RackHD](#)

RackHD is a technology stack for enabling automated hardware management and orchestration through cohesive APIs. It serves as an abstraction layer between other management layers and the underlying, vendor-specific physical hardware.

Developers can use the RackHD APIs to incorporate RackHD functionality into a larger orchestration system or to create a user interface for managing hardware services regardless of the underlying hardware in place.

The project is housed at <https://github.com/RackHD/> and available under the Apache 2.0 license (or compatible sub-licenses for library dependencies). This RackHD documentation is hosted at <http://rackhd.readthedocs.io>.

Contents

RackHD Overview

WARNING: 1.1 API DEPRECATED

RackHD serves as an abstraction layer between other M&O layers and the underlying physical hardware. Developers can use the RackHD API to create a user interface that serves as single point of access for managing hardware services regardless of the specific hardware in place.

RackHD has the ability to discover the existing hardware resources, catalog each component, and retrieve detailed telemetry information from each resource. The retrieved information can then be used to perform low-level hardware management tasks, such as BIOS configuration, OS installation, and firmware management.

RackHD sits between the other M&O layers and the underlying physical hardware devices. User interfaces at the higher M&O layers can request hardware services from RackHD. RackHD handles the details of connecting to and managing the hardware devices.

The RackHD API allows you to automate a great range of management tasks, including:

- Install, configure, and monitor bare metal hardware (compute servers, PDUs, DAEs, network switches).
- Provision and erase server OSes.
- Install and upgrade firmware.
- Monitor bare metal hardware through out-of-band management interfaces.
- Provide data feeds for alerts and raw telemetry from hardware.

Vision

Feature	Description
Discovery and Cataloging	Discovers the compute, network, and storage resources and catalogs their attributes and capabilities.
Telemetry and Genealogy	Telemetry data includes genealogical details, such as hardware, revisions, serial numbers, and date of manufacture
Device Management	Powers devices on and off. Manages the firmware, power, OS installation, and base configuration of the resources.
Configuration	Configures the hardware per application requirements. This can range from the BIOS configuration on compute devices to the port configurations in a network switch.
Provisioning	Provisions a node to support the intended application workflow, for example lays down ESXi from an image repository. Reprovisions a node to support a different workload, for example changes the ESXi platform to Bare Metal CentOS.
Firmware Management	Manages all infrastructure firmware versioning.
Logging	Log information can be retrieved for particular elements or collated into a single timeline for multiple elements within the management neighborhood.
Environmental Monitoring	Aggregates environmental data from hardware resources. The data to monitor is configurable and can include power information, component status, fan performance, and other information provided by the resource.
Fault Detection	Monitors compute and storage devices for both hard and soft faults. Performs suitable responses based on pre-defined policies.
Analytics Data	Data generated by environmental and fault monitoring can be provided to analytic tools for analysis, particularly around predictive failure.

Goals

The primary goals of RackHD are to provide REST APIs and live data feeds to enable automated solutions for managing hardware resources. The technology and architecture are built to provide a platform agnostic solution.

The combination of these services is intended to provide a REST API based service to:

- Install, configure, and monitor bare metal hardware, such as compute servers, power distribution units (PDUs), direct attached extenders (DAE) for storage, and network switches.
- Provision, erase, and reprovision a compute server's OS.
- Install and upgrade firmware for qualified hardware.
- Monitor and alert bare metal hardware through out-of-band management interfaces.
- Provide RESTful APIs for convenient access to knowledge about both common and vendor-specific hardware.
- Provide pub/sub data feeds for alerts and raw telemetry from hardware.

The RackHD Project

The original motive centered on maximizing the automation of firmware and BIOS updates in the data center, thereby reducing the extensive manual processes that are still required for these operations.

Existing open source solutions do an admirable job of inventory and bare OS provisioning, but the ability to upgrade firmware is beyond the technology stacks currently available (i.e. [xCat](#), [Cobbler](#), [Razor](#) or [Hanlon](#)). By adding an event-based workflow engine that works in conjunction with classical PXE booting, RackHD makes it possible to architect different deployment configurations as described in [how_it_works](#) and [Deployment Environment](#).

RackHD extends automation beyond simple PXE booting. It can perform highly customizable tasks on machines, as is illustrated by the following sequence:

- PXE boot the server
- Interrogate the hardware to determine if it has the correct firmware version
- If needed, flash the firmware to the correct version
- Reboot (mandated by things like BIOS and BMC flashing)
- PXE boot again
- Interrogate the hardware to ensure it has the correct firmware version.
- SCORE!

In effect, RackHD combines open source tools with a declarative, event-based workflow engine. It is similar to Razor and Hanlon in that it sets up and boots a microkernel that can perform predefined tasks. However, it extends this model by adding a remote agent that communicates with the workflow engine to *dynamically* determine the tasks to perform on the target machine, such as zero out disks, interrogate the PCI bus, or reset the IPMI settings through the hosts internal KCS channel.

Along with this agent-to-workflow integration, RackHD optimizes the path for interrogating and gathering data. It leverages existing Linux tools and parses outputs that are sent back and stored as free-form JSON data structures.

The workflow engine was extended to support polling via out-of-band interfaces in order to capture sensor information and other data that can be retrieved using IPMI. In RackHD these become *pollers* that periodically capture telemetry data from the hardware interfaces.

What RackHD Does Well

RackHD is focused on being the lowest level of automation that interrogates agnostic hardware and provisions machines with operating systems. The API can be used to pass in data through variables in the workflow configuration, so you can parameterize workflows. Since workflows also have access to all of the SKU information and other catalogs, they can be authored to react to that information.

The real power of RackHD, therefore, is that you can develop your own workflows and use the REST API to pass in dynamic configuration details. This allows you to execute a specific sequence of arbitrary tasks that satisfy your requirements.

When creating your initial workflows, it is recommended that you use the existing workflows in our code repository to see how different actions can be performed.

What RackHD Doesn't Do

RackHD is a comparatively passive system. Workflows do not contain the complex logic for functionality that is implemented in the layers above hardware management and orchestration. For example, workflows do not provide scheduling functionality or choose which machines to allocate to particular services.

We document and expose the events around the workflow engine to be utilized, extended, and incorporated into an infrastructure management system, but we did not take RackHD itself directly into the infrastructure layer.

Comparison with Other Projects

Comparison to other open source technologies:

Cobbler comparison

- Grand-daddy of open source tools to enable PXE imaging
- Original workhorse of datacenter PXE automation
- XML-RPC interface for automation, no REST interface
- No dynamic events or control for TFTP, DHCP
- Extensive manual and OS level configuration needed to utilize
- One-shot operations - not structured to change personalities (OS installed) on a target machine, or multiple reboots to support some firmware update needs
- No workflow engine or concept of orchestration with multiple reboots

Razor/Hanlon comparison

- HTTP wrapper around stock open source tools to enable PXE booting (DHCP, TFTP, HTTP)
- Razor and Hanlon extended beyond Cobbler's concepts to include microkernel to interrogate remote host and use that information with policies to choose what to PXE boot
- Razor isn't set to make dynamic responses through TFTP or DHCP where RackHD uses dynamic responses based on current state for PXE to enable workflows
- Catalog and policy are roughly equivalent to RackHD default/discovery workflow and SKU mechanism, but oriented on single OS deployment for a piece or type of hardware
- Razor and Hanlon are often focused on hardware inventory to choose and enable OS installation through Razor's policy mechanisms.
- No workflow engine or concept of orchestration with multiple reboots
- Tightly bound to and maintained by Puppet
- Forked variant [Hanlon](#) used for Chef Metal driver

xCat comparison

- HPC Cluster Centric tool focused on IBM supported hardware
- Firmware update features restricted to IBM/Lenovo proprietary hardware where firmware was made to "one-shot-update", not explicitly requiring a reboot
- Has no concept of workflow or sequencing
- Has no obvious mechanism for failure recovery
- Competing with Puppet/Chef/Ansible/cfEngine to own config management story
- Extensibility model tied exclusively to Perl code
- REST API is extremely light with focus on CLI management
- Built as a master controller of infrastructure vs an element in the process

Related Projects

- CLI
 - [Ruby CLI for RackHD](#)
- OpenStack
 - [Shovel - RackHD coordinator](#)
 - [Shovel Horizon Plugin](#)

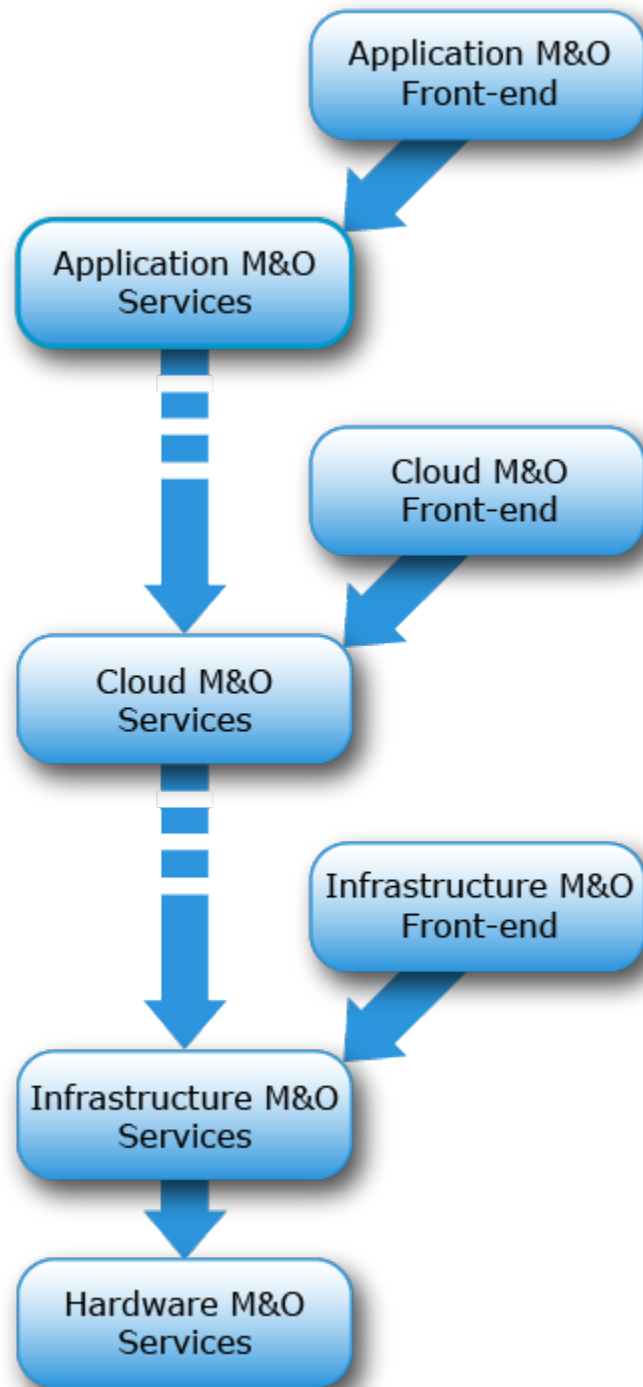
- Shovel API python client
- CloudFoundry/BOSH
 - Bosh RackHD CPI
- Docker
 - Docker Machine driver for RackHD
- Libraries
 - Golang client library to RackHD 1.1 API

System Architecture

RackHD enables much of its functionality by providing PXE boot services to machines that will be managed, and integrating the services providing the protocols used into a workflow engine. RackHD is built to download a microkernel (a small OS) crafted to run tasks in coordination with the workflow engine. The default and most commonly used microkernel is based on Linux, although WinPE and DOS network-based booting is also possible.

Theory of Operations

RackHD was born from the realization that our effective automation in computing and improving efficiencies has come from multiple layers of orchestration, each building on a lower layer. A full-featured API-driven environment that is effective spawns additional wrappers to combined the lower level pieces into patterns that are at first experimental and over time become either de facto or concrete standards.



Application automation services such as Heroku or CloudFoundry are service API layers (AWS, Google Cloud Engine, SoftLayer, OpenStack, and others) that are built overlying infrastructure. Those services, in turn, are often installed, configured, and managed by automation in the form of software configuration management: Puppet, Chef, Ansible,

etc. To automate data center rollouts, managing racks of machines, etc - these are built on automation to help roll out software onto servers - Cobbler, Razor, and now RackHD.

The closer you get to hardware, the less automated systems tend to become. Cobbler and SystemImager were mainstays of early data center management tooling. Razor (or Hanlon, depending on where you're looking) expanded on those efforts.

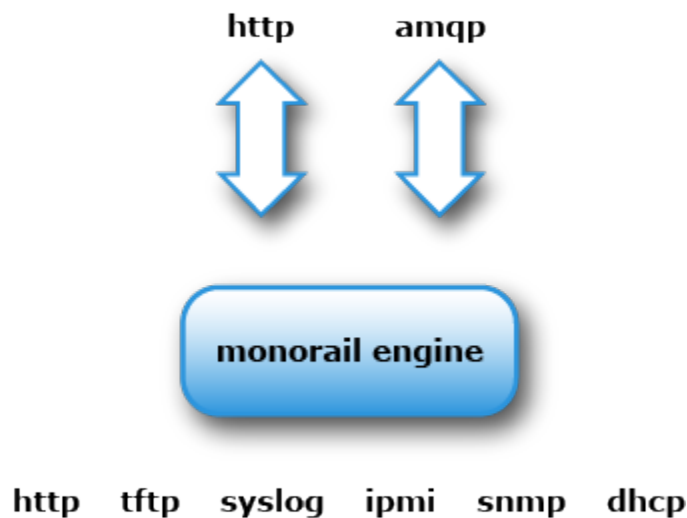
RackHD expands the capabilities of hardware management and operations beyond the mainstay features, such as PXE booting and automated installation of OS and software. It includes active metrics and telemetry, integration and annotated monitoring of underlying hardware, and firmware updating.

RackHD continues the extension by enabling automation by “playing nicely” with both existing and future potential systems, providing a consistent means of doing common automation and allowing for the specifics of various hardware vendors. It adds to existing open source efforts by providing a significant step the enablement of converged infrastructure automation.

Major Components

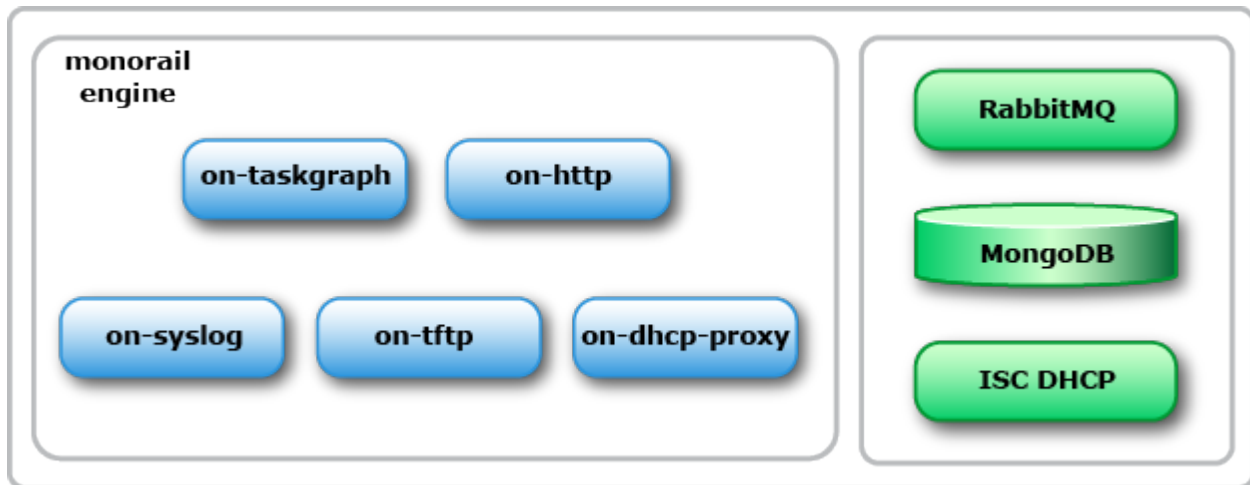
RackHD provides a REST API for the automation using an underlying workflow engine (named the “monorail engine” after a popular Seattle coffee shop: <http://www.yelp.com/biz/monorail-espresso-seattle>).

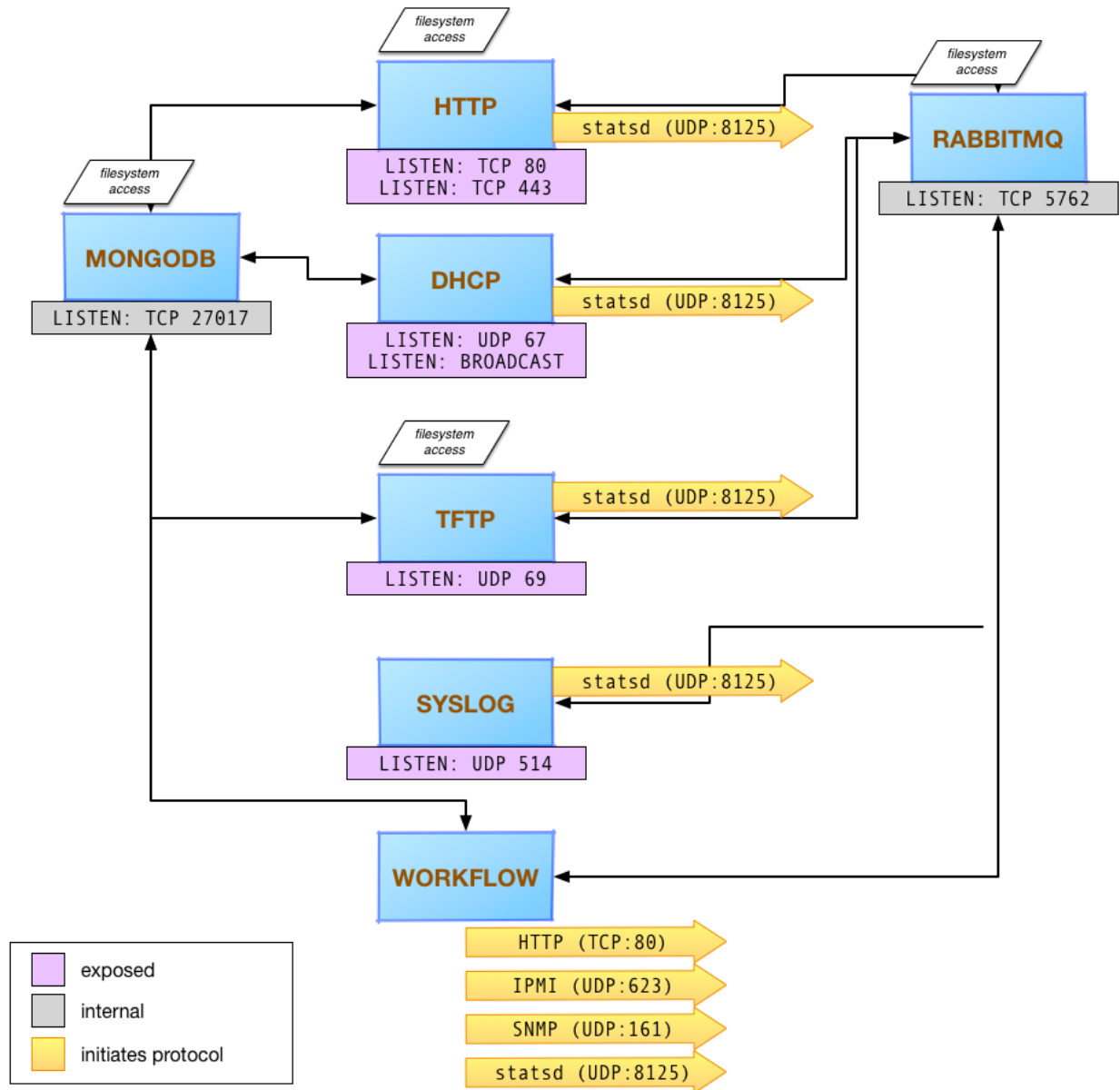
RackHD is also providing an implementation of the [Redfish specification](#) as an additional REST API to provide a common data model for representing bare metal hardware, and provides this as an aggregate for multiple back-end servers and systems.



The workflow engine operates with and coordinates with services to respond to protocols commonly used in hardware management. RackHD is structured with several independent processes, typically focused on specific function or protocol so that we can scaling or distribute them independently, using a pattern of [Microservices](#).

RackHD communicates between these using message passing over AMQP and stores data in an included persistence store. MongoDB is the default, and configurable communications layers and persistence layers are in progress.





ISC DHCP

This DHCP server provides IP addresses dynamically using the DHCP protocol. It is a critical component of a standard ‘**Preboot Execution Environment (PXE)**’ process.

on-dhcp-proxy

The DHCP protocol supports getting additional data specifically for the PXE process from a secondary service that also responds on the same network as the DHCP server. The DHCP proxy service provides that information, generated dynamically from the workflow engine.

on-tftp

TFTP is the common protocol used to initiate a PXE process. on-tftp is tied into the workflow engine to be able to dynamically provide responses based on the state of the workflow engine and to provide events to the workflow engine when servers request files via TFTP.

on-http

on-http provides both the REST interface to the workflow engine and data model APIs as well as a communication channel and potential proxy for hosting and serving files to support dynamic PXE responses. RackHD commonly uses iPXE as its initial bootloader, loading remaining files for PXE booting via HTTP and using that communications path as a mechanism to control what a remote server will do when rebooting.

on-syslog

on-syslog is a syslog receiver endpoint providing annotated and structured logging from the hosts under management. It channels all syslog data sent to the host into the workflow engine.

on-taskgraph

on-taskgraph is the workflow engine, driving actions on remote systems and processing workflows for machines being managed. Additionally, the workflow engine provides the engine for polling and monitoring.

on-taskgraph also serves as the communication channel for the microkernel to support deep hardware interrogation, firmware updates, and other actions that can only be invoked directly on the hardware (not through an out of band management channel).

Features

Bare Metal Server Automation with PXE

RackHD uses the ‘**Preboot Execution Environment (PXE)**’_ for booting and controlling servers. PXE is a vendor-independent mechanism that allows networked computers to be remotely booted and configured. PXE booting requires that **DHCP** and **TFTP** are configured and responding on the network to which the machine is attached.

RackHD uses **iPXE** as its initial bootloader. iPXE takes advantage of HTTP and permits the dynamic generation of iPXE scripts – referred to in RackHD as *profiles* – based on what the server should do when it is PXE booting.

Data center automation is enabled through each server’s **Baseboard Motherboard Controller (BMC)** embedded on the server motherboard. Using **Intelligent Platform Management Interface (IPMI)** to communicate with the BMC, RackHD can remotely power on, power off, reboot, request a PXE boot, and perform other operations.

Many open source tools, such as **Cobbler**, **Razor**, and **Hanlon** use this kind of mechanism. RackHD goes beyond this and adds a workflow engine that interacts with these existing protocols and mechanisms to let us create workflows of tasks, boot scripts, and interactions to achieve our full system automation.

The workflow engine supports RackHD responding to requests to PXE boot, like the above systems, and additionally provides an API to invoke workflows against one or more nodes. This API is intended to be used and composed into a larger system to allow RackHD to automate efforts sequences of tasks, and leverage that specifically for bare metal management. For more details on workflows, how to create them, and how to use them, please see [Workflow Graphs](#) in the [RackHD Users Guide](#).

RackHD includes defaults to automatically create and run workflows when it gets DHCP/PXE requests from a system it's never seen previously. This special case is called Discovery.

Discovery and Genealogy

RackHD supports two modes of learning about machines that it manages. We loosely group these as *passive* and *active* discovery.

- Passive discovery is where a user or outside system actively tells RackHD that the system exists. This is enabled by making a post to the REST interface that RackHD can then add to its data model.
- Active discovery is invoked when a machine attempts to PXE boot on the network that RackHD is monitoring. As a new machine PXE boots, RackHD retrieves the MAC address of the machine. If the MAC address has not been recorded, RackHD creates a new record in the data model and then invokes a default workflow. To enable active discovery, you set the default workflow that will be run when a new machine is identified to one of the discovery workflows included within the system. The most common is the SKU Discovery workflow.

For an example, the “SKU Discovery” workflow runs through its tasks as follows:

1. It runs a sub-workflow called ‘Discovery’
 - (a) Discovery is initiated by sending down the iPXE boot loader with a pre-built script to run within iPXE. This script then chainloads into a new, dynamically rendered iPXE script that interrogates the enabled network interfaces on the remote machine and reports them back to RackHD. RackHD adds this information to the machine and lookup records. RackHD then renders an additional iPXE script to be chainloaded that downloads and runs the microkernel. The microkernel boots up and requests a Node.js “bootstrap” script from RackHD. RackHD runs the bootstrap program which uses a simple REST API to “ask” what it should do on the remote host.
 - (b) The workflow engine, running the discovery workflow, provides a set of tasks to run. These tasks are matched with parsers in RackHD to understand and store the output. They work together to run Linux commands that interrogate the hardware from the microkernel running in memory. These commands include interrogating the machine’s BMC settings through IPMI, the installed PCI cards, the DMI information embedded in the BIOS, and others. The resulting information is then stored in JSON format as “catalogs” in RackHD.
 - (c) When it’s completed with all the tasks, it tells the microkernel to reboot the machine and sends an internal event that the basic bootstrapping process is finished
2. The SKU Discovery workflow then performs a workflow task process called “generate-sku” that compares the catalog data for the node against SKU definition loaded into the system through the REST interface. If the definitions match, RackHD updates its data model indicating that the node belongs to a SKU. More information on SKUs, how they’re defined, and how they can be used can be found at [SKUs](#).
3. The task “generate-enclosure” interrogates catalog data for the system serial number and/or IPMI fru devices to determine whether the node is part of an enclosure (for example, a chassis that aggregates power for multiple nodes), and updates the relations in the node document if matches are found.
4. The task “create-default-pollers” creates a set of default pollers that periodically monitor the device for system hardware alerts, built in sensor data, power status, and similar information.
5. The last task (“run-sku-graph”) checks if there are additional workflow hooks defined on the SKU definition associated with the node, and creates a new workflow dynamically if defined.

You can find the SKU Discovery graph at <https://github.com/RackHD/on-taskgraph/blob/master/lib/graphs/discovery-sku-graph.js>, and the simpler “Discovery” graph it uses at <https://github.com/RackHD/on-taskgraph/blob/master/lib/graphs/discovery-graph.js>

Notes:

- No workflow is assigned to a PXE-booting system that is already known to RackHD. Instead, the RackHD system ignores proxy DHCP requests from booting nodes with no active workflow and lets the system continue to boot as specified by its BIOS or UEFI boot order.
- The discovery workflow can be updated to do additional work or steps for the installation of RackHD, to run other workflows based on the SKU analysis, or perform other actions based on the logic embedded into the workflow itself.
- Additional pollers exist and can be configured to capture data through SNMP. The RackHD project is set up to support additional pollers as plugins that can be configured and run as desired.

Telemetry, Events and Alerting

RackHD leverages its workflow engine to also provide a mechanism to poll and collect data from systems under management, and convert that into a “live data feed”. The data is cached for API access and published through AMQP, providing a “live telemetry feed” for information collected on the remote systems.

In addition to this live feed, RackHD includes some rudimentary alerting mechanisms that compare the data collected by the pollers to regular expressions, and if they match, create an additional event that is published on an “alert” exchange in AMQP. More information can be found at [Pollers](#) in the [RackHD Users Guide](#).

RackHD also provides notification on some common tasks and workflow completion. Additional detail can be found at `rackhd/heartbeat` and `rackhd/notification`.

Additional Workflows

Other workflows can be configured and assigned to run on remote systems. For example, *OS install* can be set to explicitly power cycle (reboot) a remote node. As the system PXE boots, an installation kernel is sent down and run instead of the discovery microkernel.

The remote network-based OS installation process that runs from Linux OS distributions typically runs with a configuration file - *debseed* or *kickstart*. The monorail engine provides a means to render these configuration files through templates, with the values derived from the workflow itself - either as defaults built into the workflow, discovered data in the system (such as data within the catalogs found during machine interrogation), or even passed in as variables when the workflow was invoked by an end-user or external automation system. These “templates” can be accessed through the Monorail’s engine REST API - created, updated, or removed - to support a wide variety of responses and capabilities.

Workflows can also be chained together and the workflow engine includes simple logic (as demonstrated in the discovery workflow) to perform arbitrarily complex tasks based on the workflow definition. The workflow definitions themselves are accessible through the Monorail engine’s REST API as a “graph” of “tasks”.

For more detailed information on graphs, see the section on [Workflow Graphs](#) under our [RackHD Users Guide](#).

Workflows and tasks are fully declarative with a JSON format. A workflow task is a unit of work decorated with data and logic that allows it to be included and run within a workflow. Tasks are also mapped up “Jobs”, which is the Node.js code that RackHD runs from data included in the task declaration. Tasks can be defined to do wide-ranging operations, such as bootstrap a server node into a Linux microkernel, parse data for matches against a rule, and more.

For more detailed information on tasks, see the section on [Workflow Tasks](#) under our [RackHD Users Guide](#).

Contributing to RackHD

We certainly welcome and encourage contributions in the form of issues and pull requests, but please read the guidelines in this document before you get involved.

Since our project is relatively new, we don't yet have many hard and fast rules. As the project grows and more people get involved, we will solidify and extend our guidelines as needed.

Communicating with Other Users

We maintain a mailing list at <https://groups.google.com/d/forum/rackhd>. You can visit the group through the web page or subscribe directly by sending email to rackhd+subscribe@googlegroups.com.

We also have a #RackHD slack channel at <https://codecommunity.slack.com/messages/rackhd/>. You can receive an invite by requesting one at <http://community.codedellemc.com/>.

Submitting Contributions

To submit coding additions or changes for a repository, fork the repository and clone it locally. Then use a unique branch to make commits and send pull requests.

Keep your pull requests limited to a single issue. Make sure that the description of the pull request is clear and complete.

Run your changes against existing tests or create new ones if needed. Keep tests as simple as possible. At a minimum, make sure your changes don't break the existing project. For more information about contributing changes to RackHD, please see [Contributing Changes to RackHD](#)

After receiving the pull request, our core committers will give you feedback on your work and may request that you make further changes and resubmit the request. The core committers will handle all merges.

If you have questions about the disposition of a request, feel free to email one of our core committers.

Core Committer Team

- Michael.Hepfer@dell.com
- Andrew.Hou@dell.com
- Andre.Keedy@dell.com
- James.King@dell.com
- Lyne.Lin@dell.com
- Rahman.Muhammad@dell.com
- Jeanne.Ohren@dell.com
- Geoffrey.Reid@dell.com
- Stuart.Stanley@dell.com
- James.Turnquist@dell.com

Please direct general conversation about how to use RackHD or discussion about improvements and features to our mailing list at rackhd@googlegroups.com

Issues and Bugs

Please use <https://rackhd.atlassian.net/secure/RapidBoard.jspa?rapidView=5> to raise issues, ask questions, and report bugs.

Search existing issues to ensure that you do report a topic that has already been covered. If you have new information to share about an existing issue, add your information to the existing discussion.

When reporting problems, include the following information:

- Problem Description
- Steps to Reproduce
- Actual Results
- Expected Results
- Additional Information

Security Issues

If you discover a security issue, please report it in an email to rackhd@emc.com. Do not use the Issues section to describe a security issue.

Understanding the Repositories

The <https://github.com/rackhd/RackHD> repository acts as a single source location to help you get or build all the pieces to learn about, take advantage of, and contribute to RackHD.

A thorough understanding of the individual repositories is essential for contributing to the project. The repositories are described in our documentation at [Repositories](#).

Submitting Design Proposals

Significant feature and design proposals are expected to be proposed on the mailing list (rackhd@googlegroups.com, or at groups.google.com/forum/#!forum/rackhd) for discussion. The Core Committer team reviews the proposals to make sure architectural details are aligned, with a floating agenda updated on the RackHD Confluence page at <https://rackhd.atlassian.net/wiki/spaces/RAC1/pages/9437198/Core+Committer+Weekly+Interlock> (formerly github wiki at <https://github.com/RackHD/RackHD/wiki/Core-Committer-Meeting>). The meeting notes are posted to the google groups mailing list.

Work by dedicated teams is scheduled within a broader [RackHD Roadmap](#). External contributions are absolutely welcome outside of planning exposed in the roadmap.

Coding Guidelines

Use the same coding style as the rest of the codebase. In general, write clean code and supply meaningful and comprehensive code comments. For more detailed information about how we've set up our code, please see our [RackHD Development Guide](#).

Contributing to the Documentation

To contribute to our documentation, clone the [RackHD/docs](#) repository and submit commits and pull requests as is done for the other repositories. When we merge your pull requests, your changes are automatically published to our documentation site at <http://rackhd.readthedocs.org/en/latest/>.

Community Guidelines

This project adheres to the [Open Code of Conduct](#). By participating, you are expected to honor this code. Our community generally follows [Apache voting guidelines](#) and utilizes [lazy consensus](#) for logistical efforts.

RackHD Development Guide

Repositories

Applications

Appli- cation	Repository	Description
on-tftp	https://github.com/RackHD/on-tftp	Node.js application provided TFTP service integrated with the workflow engine. TFTP is the common protocol used to initiate a PXE process, and on-tftp is tied into the workflow engine to be able to dynamically provide responses based on the state of the workflow engine, and to provide events to the workflow engine when servers request files via TFTP
on-http	https://github.com/RackHD/on-http	Node.js application provided HTTP service integrated with the workflow engine. RackHD commonly uses iPXE as its initial bootloader, loading remaining files for PXE booting via HTTP and using that communications path as a mechanism to control what a remote server will do when rebooting. on-http also serves as the communication channel for the microkernel to support deep hardware interrogation, firmware updates, and other actions that can only be invoked directly on the hardware and not through an out of band management channel.
on-syslog	https://github.com/RackHD/on-syslog	System endpoint integrated to feed data to the workflow engine.
on-taskgraph	https://github.com/RackHD/on-taskgraph	Node.js application providing the workflow engine. It provides functionality for running encapsulated jobs/units of work via graph-based control flow mechanisms.
on-dhcp-proxy	https://github.com/RackHD/on-dhcp-proxy	Node.js application providing DHCP proxy support in the workflow engine. The DHCP protocol supports getting additional data specifically for the PXE process from a secondary service that also responds on the same network as the DHCP server. The DHCP proxy service provides that information, generated dynamically from the workflow engine.
on-wss	https://github.com/RackHD/on-wss	Node.js application providing websocket update support from RackHD for UI interactions

Libraries

Library	Repository	Description
core	https://github.com/RackHD/on-core	Core libraries in use across Node.js applications.
tasks	https://github.com/RackHD/on-tasks	Node.js task library for the workflow engine. Tasks are loaded and run by taskgraphs as needed.
redfish-client-node	https://github.com/RackHD/redfish-client-node	Node.js client library for interacting with Redfish API endpoints.

Supplemental Code

Library	Repository	Description
Web user interface	https://github.com/RackHD/web-ui	Initial web interfaces to some of the APIs - multiple interfaces embedded into a single project.
statsd	https://github.com/RackHD/statsd	A local statsD implementation that makes it easy to deploy on a local machine for aggregating and summarizing application metrics.
Image-Builder	https://github.com/RackHD/imagebuilder	Tooling to build RackHD binary files, including the microkernel docker images and specific iPXE builds
SKU Packs	https://github.com/RackHD/skupack	Example SKU pack definitions and example code
Build Config	https://github.com/RackHD/build-config	Scripts and tooling to support CI of RackHD

Documentation

Repository	Description
https://github.com/RackHD/docs	The RackHD documentation as published to http://rackhd.readthedocs.org/en/latest/ .

Repositories Status

The following badges in the tables may take a while to load.

Repository	Travis-Ci Build	Code Climate	Code Coverage
on-core			
on-dhcp-proxy			
on-http			
on-imagebuilder		N/A	N/A
on-statsd			
on-syslog			
on-taskgraph			
on-tasks			
on-tftp			
on-web-ui			N/A
on-wss			N/A

Contributing Changes to RackHD

Guidelines for merging pull requests

For code changes, we currently use a guideline of [lazy consensus](#) with two positive reviews with at least one of those reviews being one of the core maintainers and no negative votes. And of course, the gates for the pull requests must pass as well (unit tests, etc).

If you put a review up, please be explicit with a vote (+1, -1, or +/-0) so we can distinguish questions asking for information or background from reviews implying that the relevant change should not be merged. Likewise if you put up a change for review as a pull request, a -1 review comment isn't a reflection on you as a person, instead is a request to make a modification before that pull request should be merged.

For those with commit privileges

See <https://github.com/RackHD/RackHD/wiki/Merge-Guidelines> for more informal guidelines and rules of thumb to follow when making merge decisions.

Getting commit privileges

The core committer team will grant contributor rights to the RackHD project using a [lazy consensus](#) mechanism. Any of the maintainers/core contributors can nominate someone to have those privileges, and with two +1 votes and no negative votes, the team will grant commit privileges.

The core team will also be responsible for removing commit privileges when appropriate - for example for malicious merge behavior or just inactivity over an extended period of time.

Quality gates for the pull requests

There are three quality gates to ensure the pull requests quality, [Hound](#) for code style check, [Travis CI](#) for unit-test and coveralls, [Jenkins](#) for the combination test including unit-test and smoke test. When a pull request is created, all tests will run automatically, and the test results can be found in the merge status field of each pull request page. Running unit/functional tests locally prior to creating a pull request is strongly encouraged. This would hopefully minimize the amount errors seen during PR submission and lessen a dependency on Travis/Jenkins to test code before it's really ready to be submitted.

Hound

Hound works with [jshint](#) and comments on style violations in pull requests. Configuration files `.hound.yml` and `.jshintrc` have been created in each repository, so before creating a pull request, you can check code style locally with jshint to find out style violations beforehand.

Travis CI

Travis CI runs the unit tests, and then does some potentially ancillary actions. The build specifics are detailed in the `.travis.yml` file within each repository. For finding out basic errors before creating a pull request, you can run unit test locally using `npm test` within each repository.

Concourse

RackHD uses Concourse CI to monitor and perform quality gate tests on all pull requests prior to merge. The gates include running all the unit tests, running all dependent project unit tests with the code proposed from the pull request, running an integration “smoke test” to verify basic end to end functionality and commenting on the details of test case failure. Concourse can also take instructions from pull request comments or description in order to handle more complex test scenarios. Instructions can be written in the pull request description or comments.

All pull requests will need to be labeled with the “run-test” label before the quality gate tests will run. This label needs to be set by a RackHD Commit.

The following table show all the Jenkins Instructions and usage:

Instruction	Description	Detailed Usage
<code>depends on: pr1_url depends on: pr2_url ...</code>	Trigger one Jenkins test that using the commits of all interdependent pull requests.	<p>RackHD is a multi repository project, so there are times one new feature needs changes on two or more repositories. In such situation neither Concourse test for single pull request can pass. This command is order to solve this problem.</p> <p>Recommended usage: for interdependent pull requests, first create pull request one by one, but do not label any PRs with “run-test”. When creating the last pull request include the depend statements in the description:</p> <pre>depends on: pr1_url depends on: pr2_url ...</pre> <p>Then set the “run-test” label only on the pull request that includes the depends on instruction.</p> <p>The interdependent test result will be written back to all interdependent pull requests. The unit test error log will be commented on each related pull request, the functional test error log will only be commented on the main pull request, the one with the “depends on ...” instruction.</p>

Naming Conventions

Workflows

We use the following conventions when creating workflow-related JSON documents:

Tasks

For task definitions, the only convention is for values in the “injectableName” field. We tend to prefix all names with “Task.” and then add some categorization to classify what functionality the task adds.

Examples:

```
Task.Os.Install.CentOS
Task.Os.Install.Ubuntu
Task.Obm.Node.PowerOff
Task.Obm.Node.PowerOn
```

Graphs

For graph definitions, conventions are pretty much the same as tasks, except “injectableName” is prefixed by “Graph.”.

Examples:

```
Graph.Arista.Zerotouch.vEOS
Graph.Arista.Zerotouch.EOS
```


Microkernel docker image

Image Names

We tend to prefix docker images with *micro_* along with some information about which RancherOS the docker image was built off and information about what is contained within the docker image. Images are suffixed with *docker.tar.xz* because they are xz'd tar archives contain docker image.

Examples:

```
micro_1.2.0_flashupdt.docker.tar.xz
micro_1.2.0_brocade.docker.tar.xz
micro_1.2.0_all_binaries.docker.tar.xz
```

Image Files

When adding scripts and binaries to docker image, we typically put them in /opt within subdirectories based on vendor.

Examples:

```
/opt/MegaRAID/MegaCli/MegaCli64
/opt/MegaRAID/StorCli/storcli64
/opt/mpt/mpt3fusion/sas3flash
```

If you want to add binaries or scripts and reference them by name rather than their absolute paths, then add them to /usr/local/bin or any other directory in the default PATH for bash.

File Paths

Our HTTP server will serve docker images from /opt/monorail/static/http. It is recommended that you create subdirectories within this directory for further organization.

Examples:

```
/opt/monorail/static/http/teamA/intel_flashing/micro_1.2.0_flashupdt.docker.tar.xz
/opt/monorail/static/http/teamA/generic/micro_1.2.0_all_binaries.docker.tar.xz
```

These file paths can then be referenced in workflows starting from the base path of /opt/monorail/static/http, so the above paths are referenced for download as:

```
teamA/intel_flashing/micro_1.2.0_flashupdt.docker.tar.xz
teamA/generic/micro_1.2.0_all_binaries.docker.tar.xz
```

API Versioning Conventions

All current APIs are prefixed with:

```
/api/current
```

RackHD extenders can supplement the central API (common) with versioned customer-specific APIs in parallel.

Referencing API Versions in URIs

Use the following convention when referencing API version:

```
/api/current/...
/api/1.1/...
/api/1.2/...
```

The second `/[...]/` block in the URI is the version number. The “current” or “latest” placeholder points to the latest version of the API in the system.

Multiple API versions can be added in parallel. Use N, N-1, N-2, etc. as the naming convention.

All API versioning information should be conveyed in HTTP headers.

Versioning Resources

A translation and validation chain is used to support versioned “types” for URI resources from the RackHD system. The chain flow is:

BUSINESS OBJECT — TRANSLATE — VALIDATE

Data objects should be versioned in line with the API version.

API Version Guidelines

Use the following guide lines when determining if a new API version is needed.

The following changes require a new API version:

- changing the semantic meaning of a URI route
- removing a URI route

The following changes do not require a new API version:

- adding an entirely new URI route
- changing the query parameters (pagination, filtering, etc.) accepted by the URI route
- changing the return values on error conditions
- changing the data structure for a resource at a given URI

AMQP Message Bus Conventions

At the top level, we utilize 9 exchanges for passing various messages between key services and processes:

Configuration

RPC channel for making dynamic system configuration changes

Routing keys:

```
methods.set  
methods.get
```

Events

One to many broadcast of events applicable to workflows and reactions (where poller/telemetry events will be placed in the future as well)

Routing keys:

```
tftp.success.[nodeid]
tftp.failure.[nodeid]
http.response.[nodeid]
dhcp.bind.success.[nodeid]
task.finished.[taskid]
graph.started.[graphid]
graph.finished.[graphid]
sku.assigned.[nodeid]
```

HTTP

Routing keys:

```
http.response
```

(uncertain - duplicate of *http.response.[nodeid]*?)

DHCP

RPC channel for interrogating the DHCP service

Routing keys:

```
methods.lookupIpLease
methods.ipInRange
methods.peekLeaseTable
methods.removeLease
methods.removeLeaseByIp
methods.pinMac
methods.unpinMac
methods.pinIp
methods.unpinIp
```

TFTP

(nothing defined)

Logging

Routing keys:

```
emerg
alert
error
warning
notice
info
debug
silly
```

task-graph-runner

RPC mechanism for communicating with process running workflows

Routing keys:

```
methods.getTaskGraphLibrary
methods.getTaskLibrary
methods.getActiveTaskGraph
methods.getActiveTaskGraphs
methods.defineTaskGraph
methods.defineTask
methods.runTaskGraph
methods.cancelTaskGraph
methods.pauseTaskGraph
methods.resumeTaskGraph
methods.getTaskGraphProperties
```

Scheduler

RPC mechanism for scheduling tasks within a workflow to run

```
schedule
```

Task

RPC mechanism for tasks to interrogate or interact with workflows (task-graphs)

```
run.[taskid]
cancel.[taskid]
methods.requestProfile.[id] (right now, nodeId)
methods.requestProperties.[id] (right now, nodeId)
methods.requestCommands.[id] (right now, nodeId)
methods.respondCommands.[id] (right now, nodeId)
methods.getBootProfile.[nodeid]
methods.activeTaskExists.[nodeId]
methods.requestPollerCache
ipmi.command.[command].[graphid] (right now, command is 'power', 'sel' or 'sdr')
ipmi.command.[command].result.[graphid] (right now, command is 'power', 'sel' or 'sdr')
run.snmp.command.[graphid]
snmp.command.result.[graphid]
poller.alert.[graphid]
```

Messenger design notes

These are design notes from the original creation of the messenger service used by all applications in RackHD through the core libraries

The code to match these designs is available at <https://github.com/RackHD/on-core/blob/master/lib/common/messenger.js>

Messenger provides functionality to our core code for communicating via AMQP using RabbitMQ.

There are 3 main operations that are provided for communication including the following:

- Publish (Exchange, Topic, Data) -> Promise (Success)

- Subscribe (Exchange, Topic, Callback) - Promise (Subscription)
- Request (Exchange, Topic, Data) -> Promise (Response)

Within these operations we provide additional functionality for object marshaling, object validation, and tracing of requests.

Publish (Exchange, Topic, Data) -> Promise (Success)

Publish provides the mechanism to send data to a particular RabbitMQ exchange & topic.

Subscribe (Exchange, Topic, Callback) -> Promise (Subscription)

Subscribe provides the mechanism to listen for publishes or requests which are provided through the callback argument. The subscribe callback receives data in the form of the following:

```
function (data, message) {
  /*
   * data - The published message data.
   * message - A Message object with additional data and features.
   */
}
```

To respond to a message we support the Promise deferred syntax.

Success

```
message.resolve({ hello: 'world' });
```

Failure

```
message.reject(new Error('Some Error'));
```

Request (Exchange, Topic, Data) -> Promise (Response)

Request is a wrapper around the Publish/Subscribe mechanism which will first create a reply queue for a response and then publish the data to the requested exchange & topic. It's assumed that a Subscriber using the Subscribe API will respond to the message or a timeout will occur. The reply queue is automatically generated and disposed of at the end of the request so no subscriptions need to be managed by the consumer.

Object Marshaling

While plain JavaScript objects can be sent over the messenger it also supports marshaling of Serializable types in On-Core. Objects which implement the Serializable interface can be marshaled over AMQP by using a constructor initialization convention and by registering their type with the messenger. When sending a Serializable object over AMQP the messenger uses the registered type to decorate the AMQP message in a way in which a receiver can create a new copy of the object using its typed constructor. Subscribers who receive constructed types will have access to them directly through their data value in the subscriber callback.

Object Validation

On publish and on subscription callback the messenger will also validate Serializable objects using the Validatable base class. Validation is provided via JSON Schemas which are attached to the sub-classed Validatable objects. If an object to be marshaled is Validatable the messenger will validate the object prior to publish or subscribe callback.

Future versions of the messenger will support subscription and request type definitions which will allow consumers to identify what types of objects they expect to be notified about which will give the messenger an additional means of ensuring communications are handled correctly. Some example schemas are listed below: MAC Address

```
{
  id: 'MacAddress',
  type: 'object',
  properties: {
    value: {
      type: 'string',
      pattern: '^[0-9a-fA-F][0-9a-fA-F:]{5}([0-9a-fA-F][0-9a-fA-F])$'
    }
  },
  required: [ 'value' ]
}
```

IP Address

```
{
  id: 'IpAddress',
  type: 'object',
  properties: {
    value: {
      type: 'string',
      format: 'ipv4'
    }
  },
  required: [ 'value' ]
}
```

Lookup Model (via On-Http)

```
{
  id: 'Serializables.V1.Lookup',
  type: 'object',
  properties: {
    node: {
      type: 'string'
    },
    ipAddress: {
      type: 'string',
      format: 'ipv4'
    },
    macAddress: {
      type: 'string',
      pattern: '^[0-9A-Fa-f]{2}[:-]){5}([0-9A-Fa-f]{2})$'
    }
  },
  required: [ 'macAddress' ]
}
```

Additional Information

With the primary goal of the messenger being to simplify usage patterns for the consumer not all of the features have been highlighted. Below is a quick recap of the high level features.

- Publish, Subscribe, and Request/Response Patterns.
- Optional Object Marshaling.

- Optional Object Validation via JSON Schema.
- Publish & Subscribe use their own connections to improve latency in request/response patterns.
- Automatic creation of exchanges on startup.
- Automatic subscription management for Request/Response patterns.
- Automatic Request correlation and context marshaling.

Logging in RackHD

Log Levels

We have a common set of logging levels within RackHD, used across the projects and applications. The levels are defined in the [on-core library](#)

The conventions for using the levels are:

critical Used for logging terminal failures that are crashing the system, for information to support post-failure debugging. Errors logged as critical are expected to be terminal and will likely result in the application crashing or failing to start.

Errors logged at a **critical** level should be actionable in that the tracebacks or logged errors should allow resolution of the error with a code or configuration update. These errors are generally considered failures of the program to anticipate corner conditions or failure modes.

error Logging errors that may (or will) result in the application behaving in an unexpected fashion. Assertion/precondition errors are appropriate here, as well as any error that would generate an “unknown” error and be exposed via a 500 response (i.e. an undefined error) in an HTTP response code. The results of these errors are not expected to be terminal to the operation of the application.

Errors logged at an **error** level should be actionable in that the tracebacks or logged errors should allow resolution of the error with a code or configuration update. These errors are generally considered failures of the program to anticipate corner conditions or failure modes.

warning An expected error condition or fault in inputs to which the application responds correctly, but the end-user action may not be what they intended. Incorrect passwords, or actions that are not allowed because they conflict with existing configurations are appropriate for this level.

Errors logged at an **warning** level may not be actionable, but should be informative in the logs to indicate what the failure was. Errors where secure information are part of the response may include more information in logs than in a response of the end user for security considerations.

info Informational data about current execution that would be relevant to regular use of the application. Not generally considered “errors” at the log level of **info**, this level should be used judiciously with the idea that regular operation of the application is likely to run with log filtering set to allow **info** logging.

Information logged at the **info** is not expected to be actionable, but may be expected to be used in external systems collecting the log information for regular operational metrics.

debug Informational data about current execution that would be relevant to debugging or detailed analysis of the application, typically for a programmer, or to generate logs for post-analysis by a someone familiar with the code in the project. Information is not considered “errors” at the log level of **debug**.

Information logged at the **debug** is not expected to be actionable, but may be expected to be used in external systems collecting the log information for debugging or post-analysis metrics.

Setting up and using Logging

Using our dependency injection libraries, it's typical to inject `Logger` and then use it within appropriate methods. Within factory methods for services or modules, `Logger` is initialized with the module name, which annotates the logs with information about where the logs were coming from.

An example of this:

```
di.annotate(someFactory, new di.Inject('Logger'))

function someFactory (Logger) {
    var logger = Logger.initialize(someFactory);
}
```

with `logger` being used later within the relevant scope for logging. For example:

```
function foo(bar, baz) {
    logger.debug("Another request was made with ", {id: baz});
}
```

The definitions for the methods and what the code does can be found in the [logger module](#).

Deprecation

There is a special function in our logging common library for including in methods you're attempting to deprecate:

```
logger.deprecate("This shouldn't be used any longer", 2)
```

Which will generate log output at the **error** for assistance in identifying methods, APIs, or subsystems that are still in use but in the process of being deprecated for replacement.

RackHD Debugging Guide

Discovery with a Default Workflow

Sequence Diagram for the Discovery Workflow

The diagram is made with [WebSequenceDiagrams](#).

To see if the DHCP request was received by ISC DHCP, look in `/var/log/syslog` of the RackHD host. *grep DHCP /var/log/syslog* works reasonably well - you're looking for a sequence like this:

```
Jan  8 15:43:43 rackhd-demo dhclient: DHCPDISCOVER on eth0 to 255.255.255.255 port 67 interval 3 (xid=0)
Jan  8 15:43:43 rackhd-demo dhclient: DHCPREQUEST of 10.0.2.15 on eth0 to 255.255.255.255 port 67 (xid=0)
Jan  8 15:43:43 rackhd-demo dhclient: DHCPOFFER of 10.0.2.15 from 10.0.2.2
Jan  8 15:43:43 rackhd-demo dhclient: DHCPACK of 10.0.2.15 from 10.0.2.2
```

You should also see the DHCP proxy return the bootfile. In the DHCP-proxy logs, look for lines with *DHCP.messageHandler*:

```
S 2016-01-08T19:31:43.268Z [on-dhcp-proxy] [DHCP.messageHandler] [Server] Unknown node 08:00:27:f3:9c:12
```

And immediately thereafter, you should see the server request the file from TFTP:

```
S 2016-01-08T19:31:43.352Z [on-tftp] [Tftp.Server] [Server] tftp: 67.300 monorail.ipxe
```


Default discovery workflow

```

title Default Discovery Workflow
Server->RackHD: DHCP from PXE(nic or BIOS)
RackHD->Server: ISC DHCP response with IP
note over RackHD:
    If the node is already "known",
    it will only respond if there's an active workflow
    that's been invoked related to the node
end note
RackHD->Server: DHCP-proxy response with bootfile
Server->RackHD: Request to download bootfile via TFTP
RackHD->Server: TFTP sends requested file (monorail.ipxe)
note over Server:
    Server loads monorail.ipxe
    and initiates on bootloader
end note
Server->RackHD: IPXE script requests what to do from RackHD (http)
note over RackHD:
    RackHD looks up IP address of HTTP request from iPXE script to find the node via its mac-address
    1) If the node is already "known", it will only respond if there's an active workflow
    that's been invoked related to the node.
    2) If the node isn't known, it will create a workflow (default is the workflow 'Graph.Sku.Discovery')
    and respond with an iPXE script to initiate that.
end note
RackHD->Server: iPXE script (what RackHD calls a Profile) (via http)
note over Server:
    iPXE script with RancherOS vmlinuz,
    initrd and cloud-config (http)
end note
Server->RackHD: iPXE requests static file - the RancherOS vmlinuz kernel
RackHD->Server: RancherOS vmlinuz (http)
Server->RackHD: iPXE requests static file - RancherOS initrd
RackHD->Server: RancherOS initrd (http)
note over Server:
    Server loads the vmlinuz and initrd,
    and transfers control (boots RancherOS)
end note
Server->RackHD: RancherOS requests cloud-config - RancherOS cloud-config
RackHD->Server: RancherOS cloud-config(http)
Server->RackHD: RancherOS loads discovery docker image from Server
note over Server:
    the discovery container is set to request
    and launch a NodeJS task runner
end note
Server->RackHD: requests the bootstrap.js template
RackHD->Server: bootstrap.js filled out with values specific to the node based on a lookup
note over Server:
    runs node bootstrap.js
end note
Server->RackHD: bootstrap asks for tasks (what should I do?)
RackHD->Server: data packet of tasks (via http)
note over Server:
    Discovery Workflow
    passes down tasks to
    interrogate hardware
end note
loop for each Task from RackHD

```

```

Server->RackHD: output of task
end
note over RackHD
    Task output stored as catalogs in RackHD related to the node.
    If RackHD is configured with SKU definitions,
    it processes these catalogs to determine the SKU.
    If there's a SKU specific workflow defined, control is continued to that.
    The discovery workflow will create an enclosure node based on the catalog data.
    The discovery workflow will also create IPMI pollers for the node,
    if relevent information can be found in the catalog.
    The discovery workflow will also generate tag for the node,
    based on user-defined tagging rules.
end note
Server->RackHD: bootstrap asks for tasks (what should I do?)
RackHD->Server: Nothing more, thanks - please reboot (via http)

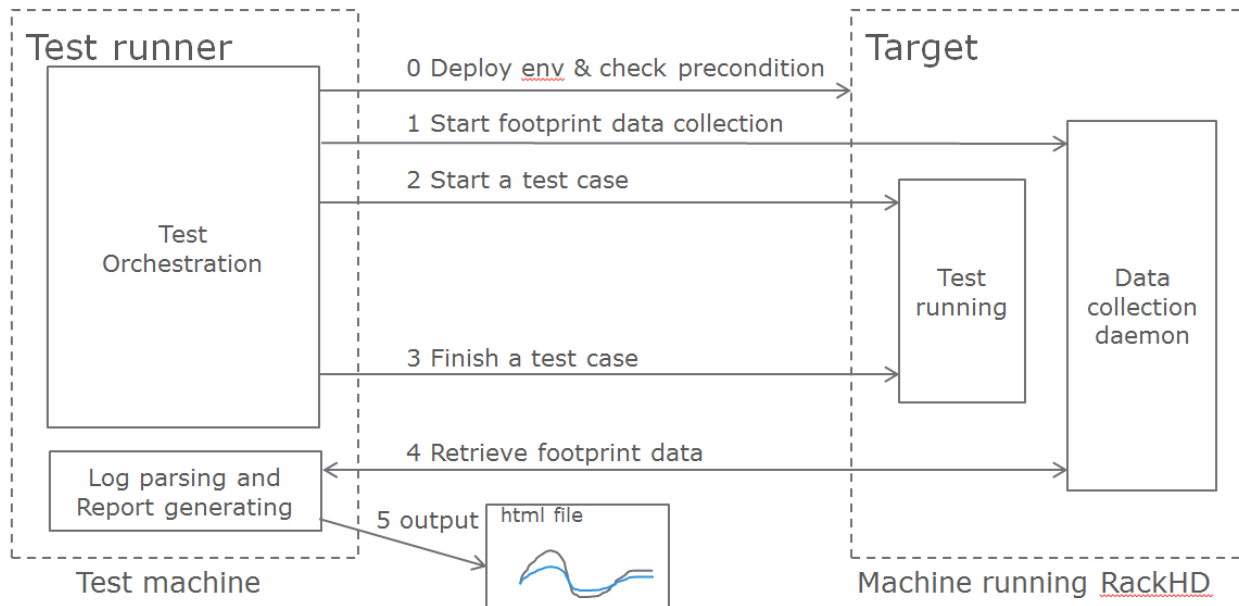
```

Footprint Benchmark Test

Footprint benchmark test collects system data when running poller (15min), node discovery and CentOS bootstrap test cases. It can also run independently from any test cases, allowing users to measure footprint about any operations they carry out. The data includes CPU, memory, disk and network consumption of every process in RackHD, as well as RabbitMQ and MongoDB processes. The result is presented as HTML files. For more details, please check the wiki page [proposal-footprint-benchmarks](#).

How It Works

Footprint benchmark test is integrated into RackHD test framework. It can be executed as long as the machine running the test can access the RackHD API and manipulate the RackHD machine via SSH.



Prerequisites

- The machine running RackHD can use apt-get to install packages, which means it must have accessible sources.list.
- In RackHD, compute nodes have been discovered, and pollers are running.
- No external AMQP queue with the name “graph.finished” is subscribed to RackHD, since the benchmark test uses this queue.
- Make sure the AMQP port in RackHD machine can be accessed by the test machine. If RackHD is not running in Vagrant, user can tunnel the port using the following command in RackHD machine.

```
sudo socat -d -d TCP4-LISTEN:55672,reuseaddr,fork TCP4:localhost:5672
```

How to Run

Clone the test repo from GitHub

```
git clone https://github.com/RackHD/RackHD.git
```

Enter test directory and install required modules in virtual env

```
cd RackHD/test
virtualenv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

Configure RackHD related parameters in config.ini

```
vim config/config.ini
```

Run the test. The first time user kicks off the test, he/she will be asked to input sudoer’s username and password of localhost.

```
python benchmark.py
```

If user would like to run only one of the three benchmark cases, the following command can be used

```
python benchmark.py --group=poller|discovery|bootstrap
```

Run footprint data collection independently

```
python benchmark.py --start|stop
```

To get the directory of the latest log file

```
python benchmark.py --getdir
```

After the test finishes, the result is in ~/benchmark, and arranged by the timestamp and case name. Please use the command below to open Chrome

```
chrome.exe --user-data-dir="C:/Chrome dev session" --allow-file-access-from-files
```

In the “report” directory of the case, drag the summary.html into Chrome. The footprint data and graph will be shown in the page, and user can also compare it with previous runs by selecting another case from the drop-down menu in the page.

Logged warnings FAQ

Question:

I'm seeing this warning appear in the logs but it all seems to be working. What's happening?

```
W 2016-01-29T21:06:22.756Z [on-tftp] [Tftp.Server] [Server] Tftp error
-> /lib/server.js:57
file:      monorail.ipxe
remoteAddress: 172.31.128.5
remotePort: 2070
W 2016-01-29T21:12:43.783Z [on-tftp] [Tftp.Server] [Server] Tftp error
-> /lib/server.js:57
file:      monorail.ipxe
remoteAddress: 172.31.128.5
remotePort: 2070
```

Answer:

What I learned (so I may be wrong here, but think it's accurate) is that during the boot loading/PXE process the NICs will attempt to interact with TFTP in such a way that the first request almost always fails - it's how the C code in those nics is negotiating for talking with TFTP. So you'll frequently see those errors in the logs, and then immediately also see the same file downloading on the second request from the nic (or host) doing the bootloading.

Question:

When we're bootstrapping a node (or running a workflow against a node in general) with a NUC, we sometimes see these extended messages on the server's console reading *Link..... down*, and depending on the network configuration can see failures for the node to bootstrap and respond to PXE.

Answer:

The link down is a pernicious problem for PXE booting in general, and a part of the game that's buried into how switches react and bring up and down ports. We've generally encouraged settings like "portfast" which more aggressively bring up links that are going down and coming back up with a power cycle. In the NUCs you're using, you'll see that extensively, but it happens on all networks. If you have spanning-tree enabled, some things like that - it'll expand the time. There's only so much we can do to work around it, but fundamentally it means that while the relevant computer things are "UP and OK" and has started a TFTP/PXE boot process, the switch hasn't brought the NIC link up. So we added an explicit sleep in there in the monorail.ipxe to extend 'the time to let networks converge so that the process has a better chance of succeeding.

WARNING: 1.1 API DEPRECATED

RackHD Users Guide

Deployment Environment

RackHD can use a number of different mechanisms to coordinate and control bare metal hardware, and in the most common cases, a deployment is working with at least two networks, connected on different network interface cards, to the RackHD instance.

RackHD can be configured to work with a single network, or several more networks, depending on the needs of the installation. The key elements to designing a RackHD installation are:

- understanding what network *security constraints* you are using
- understanding the *hardware controls* you're managing and how it can be configured

- understanding where and how *IP address management* is to be handled in each of the networks that the first two items mandate.

```
rackhd/../../static/vagrant
```

At a minimum, RackHD expects a “southbound” network, where it is interacting with the machines it is PXE booting a network provided with DHCP, TFTP, and HTTP and a “northbound” network where RackHD exposes the APIs for automation and interaction. This basic setup was created to allow and encourage separation of traffic for PXE booting nodes and API controls. The example setup in `../getting_started` shows a minimal configuration.

Security Constraints

RackHD as a technology is configured to control and automate hardware, which implies a number of natural security concerns. As a service, it provides an API control endpoint, which in turn uses protocols on networks relevant to the hardware it’s managing. One of the most common of those protocols is **IPMI**, which has [known security flaws](#), but is used because it’s one of the most common mechanisms to control datacenter servers.

A relatively common requirement in datacenters is that networks used for IPMI traffic are isolated from other networks, to limit the vectors by which IPMI endpoints could be attacked. When RackHD is using IPMI, it simply needs to have L3 (routed IP) network traffic to the relevant endpoints in order for the workflow engine and various controls to operate.

Access to IPMI endpoints on hardware can be separated off onto it’s own network, or combined with other networks. It is generally considered best practice to separate this network entirely, or constrain it to highly controlled networks where access is strictly limited.

Hardware Controls

KCS and controlling the BMC

Most Intel servers with BMCs include a “KCS” (Keyboard Controller Style) communications channel between the motherboard and the BMC. This allows communications between the motherboard and the BMC, where the software running on the main computer can interrogate and configure the BMC.

Software tools such as **IPMITool** on Linux can leverage this interface, which shows up as a kernel device.

RackHD is configured to use and leverage this interface by default to interrogate the BMC and provide information about it’s settings to RackHD. It can also be used by workflows set values for the BMC. If the server you are working with does not have a BMC or does not have a KCS channel (as is the case with a virtual machine), then you will often see an error message on the console of the managed server:

```
insmod: ERROR: could not load module /opt/drivers/ipmi_msghandler.ks: No such file or directory
```

when running the default RackHD discovery through the microkernel.

RackHD manages hardware generally using at least one network interface. Network switches typically have an administrator network interface, and Smart PDUs that can be managed by RackHD have a administrative gateway.

Compute servers have the most varied and complex setup, with data center servers often leveraging a BMC (Baseboard Management Controller). A BMC is a separate embedded computer monitoring and controlling a larger computer. The protocol used most commonly to communicate to a BMC is **IPMI**, the details of which can matter significantly.

Desktop class machines (and many laptops) often do not have BMCs, although some Intel desktops may have an alternative technology: [AMT](#) which provides some similar mechanisms.

You can view a detailed diagram of the components inside a BMC at [IPMI Basics](#), although every hardware vendor is slightly different in how they configure their servers. The primary difference for most Intel-based server vendors is how the BMC network interface is exposed. There are two options that you will commonly see:

- **LOM** [Lights out Management] The BMC has a dedicated network interface to the BMC
- **SOM** [“Shared on motherboard”] The network interface to the BMC shares a network interface with the motherboard. In these cases, the same physical plug is backed by two internal network interfaces (each with its own hardware address).

If you’re working with a server with a network interface shared by the motherboard and BMC, then separating the networks that provide IPMI access and the networks that the server will use during operation may be significantly challenging.

The BMC provides a lot of information about the computer, but not everything. Frequently devices such as additional NIC cards, RAID array controllers, or other devices attached to internal PCI busses aren’t accessible or known about from the BMC. This is why RackHD’s default discovery mechanism operates by [Discovery and Genealogy](#), which loads an OS into RAM on the server and uses that OS to interrogate the hardware.

IP Address Management

With multiple networks in use with RackHD, how machines are getting IP addresses and what systems are responsible for providing those IP addresses is another critical concern. Running DHCP, which RackHD integrates with tightly to enable PXE booting of hosts, must be done carefully and there should only ever be a single DHCP server running on a given layer-2 network. Many existing systems will often already have DHCP servers operational or a part of their environment, or may mandate that IP addresses are set statically or provided via a static configuration.

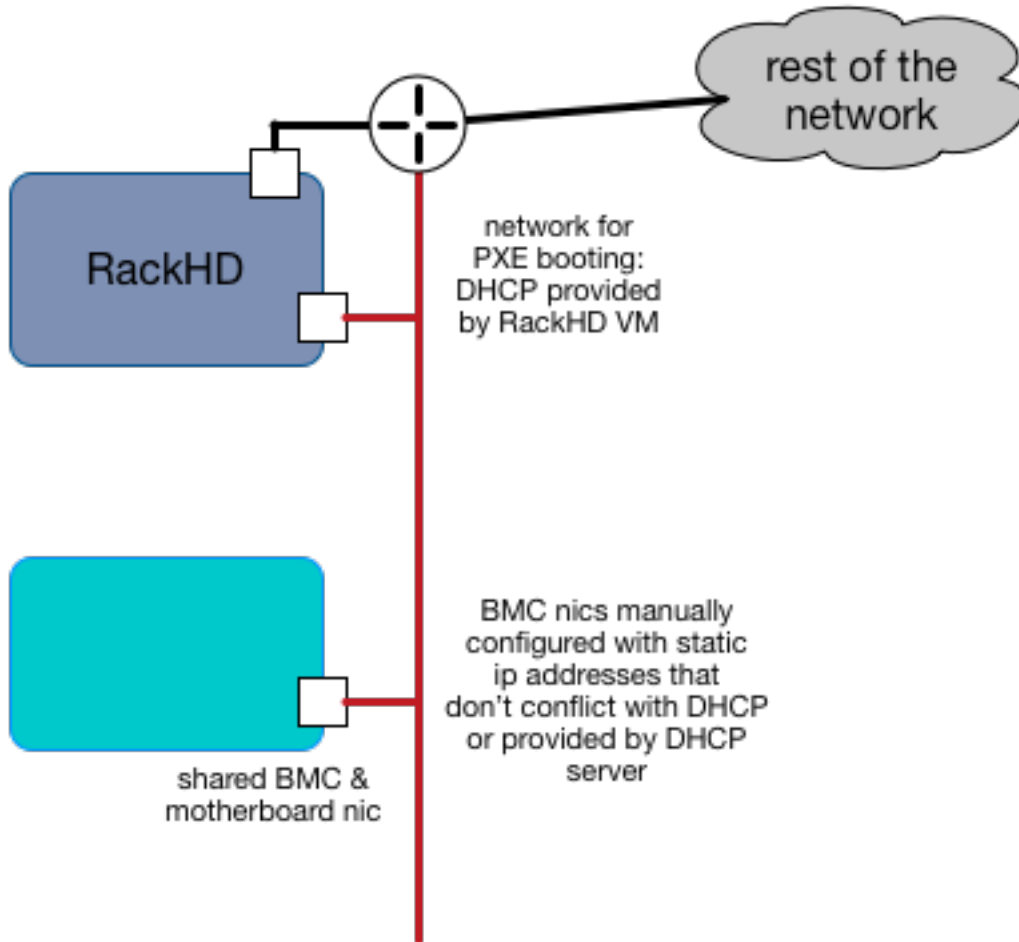
RackHD can be configured without a local DHCP instance, although DHCP is a required component for PXE booting a host. If DHCP is provided externally, then RackHD only needs to provide the *on-dhcp-proxy* process, which will need to be on the same network as the DHCP server, and leverages the DHCP protocol’s capability to separate out the service providing the TFTP boot information from the service providing IP address (and other) configuration details for hosts.

RackHD Network Access Requirements

- **DHCP-proxy** The DHCP proxy service for RackHD needs to be on the same Layer 2 (broadcast) network as DHCP to provide PXE capabilities to machines PXE booting on that network.
- **TFTP, HTTP** The PXE network also needs to be configured to expose the *south-bound* HTTP API interfaces from on-http and the on-tftp service to support RackHD PXE booting hosts by providing the bootloaders, and responding to requests for files and custom templates or scripts that coordinate with RackHD’s workflow engine.
- **IPMI, HTTP/Redfish, SNMP** Layer 3 (routed IP) access to the out of band network - the network used to communicate with server BMCs, SmartPDU management gateways, or Network switch administrative network interfaces.

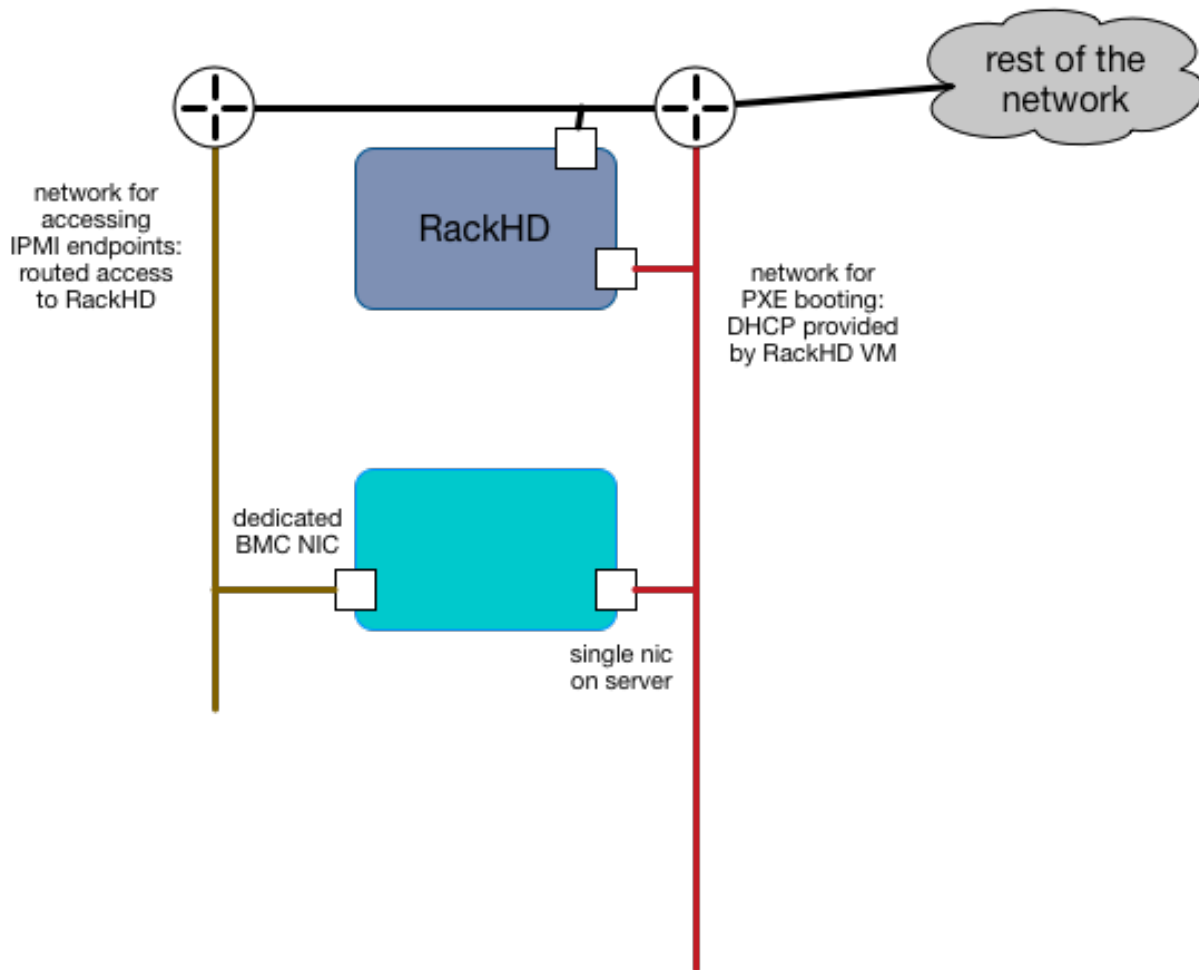
Possible Configurations

In an environment where the hardware you’re managing doesn’t have additional network interfaces, and the BMC shares the motherboard physical network interface, the configuration will be fairly limited.



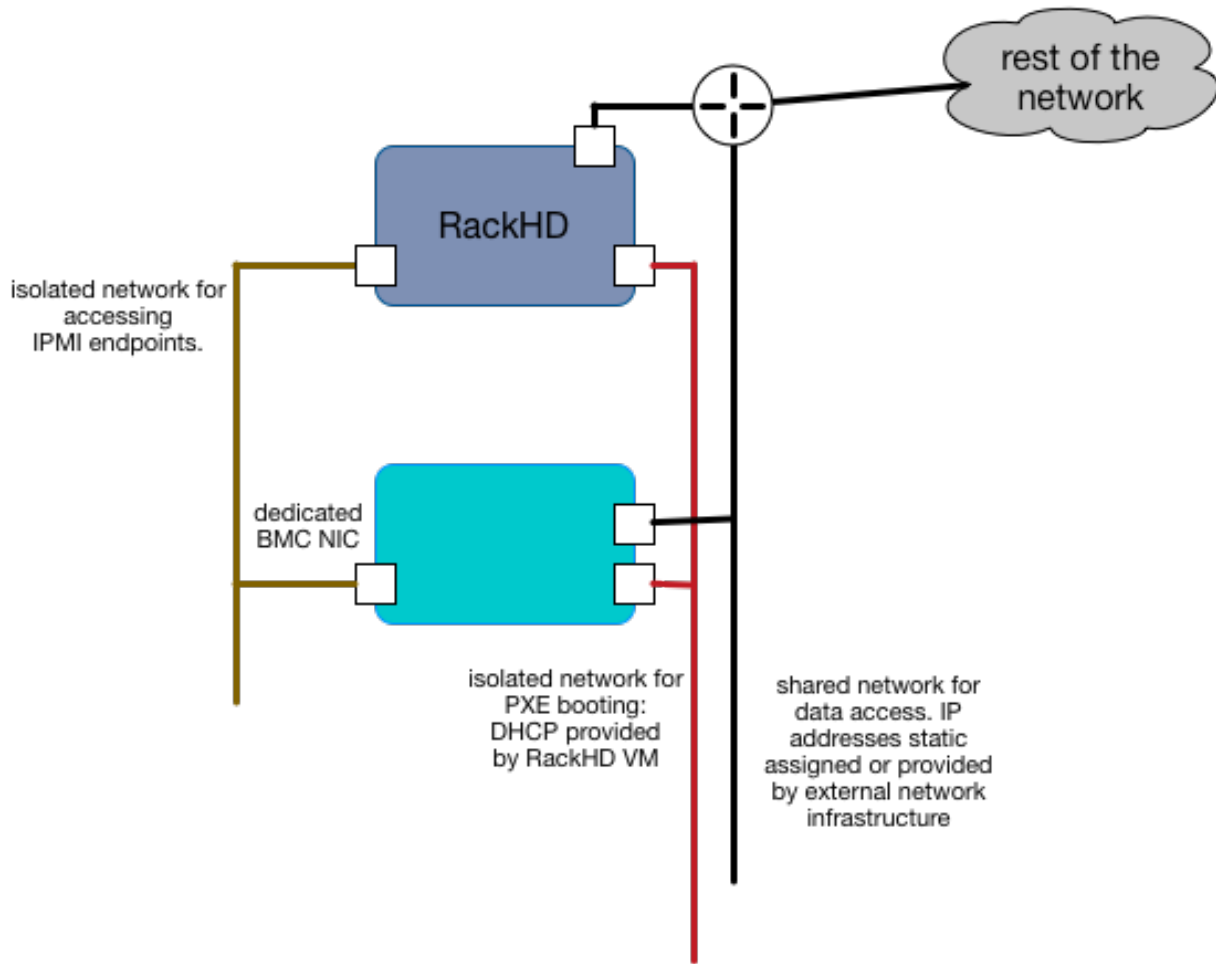
In this example, RackHD is providing DHCP to a network which is connected through a layer3 switch or router to the rest of the network. RackHD's DHCP server can provide IP addresses to the motherboard NICs as the PXE boot, and may also provide IP addresses to the BMCs if they are configured to use DHCP.

If the compute servers are not configured to use DHCP in this setup, then the BMC IP addresses must be statically set/assigned and carefully managed so as to not overlap with the DHCP range that RackHD's DHCP services are providing.



In this example, the servers have a dedicated “lights out” network interface, which is on a separate network and RackHD can access it via one of its interfaces. RackHD is still providing DHCP to the servers for PXE booting on the motherboard, but the IP addresses of the BMCs can be completely independent in how they are provided.

This example, or a variation on it, is how you might configure a RackHD deployment in a dedicated data center where the same people responsible for running RackHD are responsible for the IP addresses and general datacenter infrastructure. In general, this kind of configuration is what you might do with shared responsibilities and close coordination between network configurations within and external to RackHD



In this example, all the networks are isolated and separate, and in this case isolated to the instance of RackHD as well. RackHD may be multiple network interfaces assigned to it with various network configurations. The BMC network can be set to use a DHCP or statically assigned IP addresses - as long as the network routing is clear and consistent to RackHD. The servers also have multiple network interface cards attached to the motherboard, each of which can be on separate networks, or they can be used in combined configurations.

This example highlights how RackHD might be configured if it was being used to independently manage a rack of gear, as in an “rack of machines as an appliance” use case, or in a very large scale environment, where every rack has it’s own dedicated management network that are functionally identical.

Installation

Source Installation on Ubuntu

NICs

Ubuntu 14.04

Start with an Ubuntu trusty(14.04) instance with 2 nics:

- eth0 for the ‘public’ network - providing access to RackHD APIs, and providing routed (layer3) access to out of band network for machines under management
- eth1 for dhcp/pxe to boot/configure the machines

edit the network:

- eth0 - assign IP address as appropriate for the environment, or you can use DHCP
 - eth1 static (172.31.128.0/22)
-

Ubuntu 16.04

Start with an Ubuntu xenial(16.04) instance with 2 nics:

- ens160 for the ‘public’ network - providing access to RackHD APIs, and providing routed (layer3) access to out of band network for machines under management
- ens192 for dhcp/pxe to boot/configure the machines

edit the network:

- ens160 - assign IP address as appropriate for the environment, or you can use DHCP
- ens192 static (172.31.128.0/22)

If you start with Ubuntu xenial(16.04), please check the network config file: */etc/network/interfaces*. The ens192’s ip address is 172.31.128.1. Like as follows:

```
auto ens192
iface ens192 inet static
    address 172.31.128.1
    post-up ifconfig ens192 promisc
```

We will leverage the ansible roles created for the RackHD demonstration environment.

```
cd ~
sudo apt-get install git
sudo apt-get update
sudo apt-get dist-upgrade
sudo reboot

cd ~
git clone https://github.com/rackhd/rackhd
sudo apt-get install ansible
cd ~/rackhd/packer/ansible
ansible-playbook -i "local," -K -c local rackhd_local.yml
```

This created the default configuration file at */opt/monorail/config.json* from <https://github.com/RackHD/RackHD/blob/master/packer/ansible/roles/monorail/files/config.json>. You may need to update this and */etc/dhcpd.conf* to match your local network configuration.

This will install all the relevant dependencies and code into *~/src*, expecting that it will be run with *pm2*.

Start RackHD

```
cd ~
sudo pm2 start rackhd-pm2-config.yml
```

Some useful commands of *pm2*:

```

sudo pm2 restart all           # restart all RackHD services
sudo pm2 restart on-taskgraph  # restart the on-taskgraph service only.
sudo pm2 logs                  # show the combined real-time log for all RackHD services
sudo pm2 logs on-taskgraph     # show the on-taskgraph real-time log
sudo pm2 flush                  # clean the RackHD logs
sudo pm2 status                 # show the status of RackHD services

```

Notesisc-dhcp-server is installed through ansible playbook, but sometimes it won't start on Ubuntu boot (<https://ubuntuforums.org/showthread.php?t=2068111>), check if DHCP service is started:

```
sudo service --status-all
```

If isc-dhcp-server is not running, run below to start DHCP service:

```
sudo service isc-dhcp-server start
```

How to update to the latest code

```

cd ~/src
./scripts/clean_all.bash && ./scripts/reset_submodules.bash && ./scripts/link_install_locally.bash

```

How to Reset the Database

```
echo "db.dropDatabase()" | mongo pxe
```

Package Installation on Ubuntu

Prerequisites

NICs Ubuntu 14.04

Start with an Ubuntu trusty(14.04) instance with 2 nics:

- eth0 for the 'public' network - providing access to RackHD APIs, and providing routed (layer3) access to out of band network for machines under management
- eth1 for dhcp/pxe to boot/configure the machines

edit the network:

- eth0 - assign IP address as appropriate for the environment, or you can use DHCP
- eth1 static (172.31.128.0/22)
this is the 'default'. it can be changed, but more than one file needs to be changed.)

Ubuntu 16.04

Start with an Ubuntu xenial(16.04) instance with 2 nics:

- ens160 for the 'public' network - providing access to RackHD APIs, and providing routed (layer3) access to out of band network for machines under management
- ens192 for dhcp/pxe to boot/configure the machines

edit the network:

- ens160 - assign IP address as appropriate for the environment, or you can use DHCP

- ens192 static (172.31.128.0/22)

this is the 'default'. it can be changed, but more than one file needs to be changed.)

NodeJS 4.x If Node.js is not installed

If Node.js is installed via apt, but is older than version 4.x, do this first (apt-get installs v0.10 by default)

```
sudo apt-get remove nodejs nodejs-legacy
```

Add the NodeSource key and repository (*instructions copied from <https://github.com/nodesource/distributions#manual-installation>*):

```
curl --silent https://deb.nodesource.com/gpgkey/nodesource.gpg.key | sudo apt-key add -
VERSION=node_4.x
DISTRO="$(lsb_release -s -c)"
echo "deb https://deb.nodesource.com/$VERSION $DISTRO main" | sudo tee /etc/apt/sources.list.d/nodesource.list
echo "deb-src https://deb.nodesource.com/$VERSION $DISTRO main" | sudo tee -a /etc/apt/sources.list.d/nodesource.list

sudo apt-get update
sudo apt-get install nodejs
```

Ensure Node.js is at version 4.x, example:

```
$ node -v
v4.4.5
```

Install & Configure RackHD

After Prerequisites installation, there're two options to install and configure RackHD from package

Either (a) or (b) can lead the way to install RackHD from debian packages.

1. *Install/Configure with Ansible Playbook*
2. *Install/Configure with Step by Step Guide*

Install/Configure with Ansible Playbook (1). install git and ansible

```
sudo apt-get install git
sudo apt-get install ansible
```

(2). clone RackHD code

```
git clone https://github.com/RackHD/RackHD.git
```

The services files in /etc/init/ all need a conf file to exist in /etc/default/{service} Touch those files to allow the upstart scripts to start automatically.

```
for service in $(echo "on-dhcp-proxy on-http on-tftp on-syslog on-taskgraph");
do sudo touch /etc/default/$service;
done
```

(3). Run the ansible playbooks

These will install the prerequisite packages, install the RackHD debian packages, and copy default configuration files

```
cd RackHD/packer/ansible
ansible-playbook -c local -i "local," rackhd_package.yml
```

(4). Verify RackHD services

All the services are started and have logs in /var/log/rackhd. Verify with `service on-[something] status`. `Notesisc-dhcp-server` is installed through ansible playbook, but sometimes it won't start on Ubuntu boot (<https://ubuntuforums.org/showthread.php?t=2068111>), check if DHCP service is started:

```
sudo service --status-all
```

If `isc-dhcp-server` is not running, run below to start DHCP service:

```
sudo service isc-dhcp-server start
```

Install/Configure with Step by Step Guide (1). Install the prerequisite packages:

```
sudo apt-get install rabbitmq-server
sudo apt-get install mongodb
sudo apt-get install snmp
sudo apt-get install ipmitool

sudo apt-get install ansible
sudo apt-get install apt-mirror
sudo apt-get install amttterm

sudo apt-get install isc-dhcp-server
```

Note: MongoDB versions 2.4.9 (on Ubuntu 14.04), 2.6.10 (on Ubuntu 16.04) and 3.4.9 (on both Ubuntu 14.04 and 16.04) are verified with RackHD. For more details on how to install MongoDB 3.4.9, please refer to: <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/>

(2). Set up the RackHD bintray repository for use within this instance of Ubuntu

```
echo "deb https://dl.bintray.com/rackhd/debian trusty main" | sudo tee -a /etc/apt/sources.list
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 379CE192D401AB61
sudo apt-get update
```

(3). Install RackHD debian package

The services files in /etc/init/ all need a conf file to exist in /etc/default/{service} Touch those files to allow the upstart scripts to start automatically.

```
for service in $(echo "on-dhcp-proxy on-http on-tftp on-syslog on-taskgraph");
do sudo touch /etc/default/$service;
done
```

Install the RackHD Packages. Note: these packages are rebuilt on every commit to master and are not explicitly versioned, but intended as a means to install or update to the latest code most conveniently.

```
sudo apt-get install on-dhcp-proxy on-http on-taskgraph
sudo apt-get install on-tftp on-syslog
```

(4). Basic RackHD Configuration

DHCP

Update `dhcpd.conf` per your network configuration

```
# RackHD added lines
deny duplicates;

ignore-client-uids true;
```

```
subnet 172.31.128.0 netmask 255.255.240.0 {
    range 172.31.128.2 172.31.143.254;
    # Use this option to signal to the PXE client that we are doing proxy DHCP
    option vendor-class-identifier "PXEClient";
}
```

Notessometimes `isc-dhcp-server` won't start on Ubuntu boot (<https://ubuntuforums.org/showthread.php?t=2068111>), check if DHCP service is started:

```
sudo service --status-all
```

If `isc-dhcp-server` is not running, run below to start DHCP service:

```
sudo service isc-dhcp-server start
```

RACKHD APPLICATIONS

Create the required file `/opt/monorail/config.json`, you can use the demonstration configuration file at <https://github.com/RackHD/RackHD/blob/master/packer/ansible/roles/monorail/files/config.json> as a reference.

RACKHD BINARY SUPPORT FILES

Downloaded binary files from bintray.com/rackhd/binary and placed them using <https://github.com/RackHD/RackHD/blob/master/packer/ansible/roles/images/tasks/main.yml> as a guide.

```
#!/bin/bash

mkdir -p /var/renasar/on-tftp/static/tftp
cd /var/renasar/on-tftp/static/tftp

for file in $(echo "\
monorail.ipxe \
monorail-undionly.kpxe \
monorail-efi64-snponly.efi \
monorail-efi32-snponly.efi");do
    wget "https://dl.bintray.com/rackhd/binary/ipxe/$file"
done

mkdir -p /var/renasar/on-http/static/http/common
cd /var/renasar/on-http/static/http/common

for file in $(echo "\
discovery.docker.tar.xz \
initrd-1.2.0-rancher \
vmlinuz-1.2.0-rancher");do
    wget "https://dl.bintray.com/rackhd/binary/builds/$file"
done
```

All the services are started and have logs in `/var/log/rackhd`. Verify with `service on-[something] status`

How to Erase the Database to Restart Everything

```
sudo service on-http stop
sudo service on-dhcp-proxy stop
sudo service on-syslog stop
sudo service on-taskgraph stop
sudo service on-tftp stop

mongo pxe
  db.dropDatabase()
  ^D

sudo service on-http start
sudo service on-dhcp-proxy start
sudo service on-syslog start
sudo service on-taskgraph start
sudo service on-tftp start
```

NPM Installation

Ubuntu

Prerequisites NICs

- **Ubuntu 14.04**

1. Start with an Ubuntu trusty(14.04) instance with 2 nics:
 - eth0 for the 'public' network - providing access to RackHD APIs, and providing routed (layer3) access to out of band network for machines under management
 - eth1 for dhcp/pxe to boot/configure the machines
2. Edit the network:
 - eth0 - assign IP address as appropriate for the environment, or you can use DHCP
 - eth1 static (172.31.128.0/22)
this is the 'default'. it can be changed, but more than one file needs to be changed.)

- **Ubuntu 16.04**

1. Start with an Ubuntu xenial(16.04) instance with 2 nics:
 - ens160 for the 'public' network - providing access to RackHD APIs, and providing routed (layer3) access to out of band network for machines under management
 - ens192 for dhcp/pxe to boot/configure the machines
2. Edit the network:
 - ens160 - assign IP address as appropriate for the environment, or you can use DHCP
 - ens192 static (172.31.128.0/22)
this is the 'default'. it can be changed, but more than one file needs to be changed.)

Packages

- **NodeJS 4.x**

1. Remove Node.js (< 4.0)

If Node.js is installed via apt, but is older than version 4.x, do this first (apt-get installs v0.10 by default)

```
sudo apt-get remove nodejs nodejs-legacy
```

2. Install Node.js 4.x

Add the NodeSource key and repository (*instructions copied from <https://github.com/nodesource/distributions#manual-installation>*):

```
curl --silent https://deb.nodesource.com/gpgkey/nodesource.gpg.key | sudo apt-key add -
VERSION=node_4.x
DISTRO="$(lsb_release -s -c)"
echo "deb https://deb.nodesource.com/$VERSION $DISTRO main" | sudo tee /etc/apt/sources.list
echo "deb-src https://deb.nodesource.com/$VERSION $DISTRO main" | sudo tee -a /etc/apt/sources.list

sudo apt-get update
sudo apt-get install nodejs
```

3. Verify Node.js 4.x

```
$ node -v
v4.4.5
```

• Dependencies

Install dependency packages

```
sudo apt-get install build-essential
sudo apt-get install libkrb5-dev
sudo apt-get install rabbitmq-server
sudo apt-get install mongodb
sudo apt-get install snmp
sudo apt-get install ipmitool

sudo apt-get install git
sudo apt-get install unzip
sudo apt-get install ansible
sudo apt-get install apt-mirror
sudo apt-get install amterm

sudo apt-get install isc-dhcp-server
```

Note: MongoDB versions 2.4.9 (on Ubuntu 14.04), 2.6.10 (on Ubuntu 16.04) and 3.4.9 (on both Ubuntu 14.04 and 16.04) are verified with RackHD. For more details on how to install MongoDB 3.4.9, please refer to: <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/>

Install & Configure RackHD

1. Install RackHD NPM Packages

Install the latest release of RackHD

```
for service in $(echo "on-dhcp-proxy on-http on-tftp on-syslog on-taskgraph");
do
  npm install $service;
done
```


2. Basic RackHD Configuration

- **DHCP**

Update /etc/dhcp/dhcpd.conf per your network configuration

```
# RackHD added lines
deny duplicates;

ignore-client-uids true;

subnet 172.31.128.0 netmask 255.255.240.0 {
    range 172.31.128.2 172.31.143.254;
    # Use this option to signal to the PXE client that we are doing proxy DHCP
    option vendor-class-identifier "PXEClient";
}
```

- **Open Ports in Firewall**

If the firewall is enabled, open below ports in firewall:

- 4011/udp
- 8080/tcp
- 67/udp
- 8443/tcp
- 69/udp
- 9080/tcp

An example of opening port:

```
sudo ufw allow 8080
```

- **CONFIGURATION FILE**

Create the required file /opt/monorail/config.json , you can use the demonstration configuration file at <https://github.com/RackHD/RackHD/blob/master/packer/ansible/roles/monorail/files/config.json> as a reference.

- **RACKHD BINARY SUPPORT FILES**

Download binary files from bintray and placed them with below shell script.

```
#!/bin/bash

mkdir -p node_modules/on-tftp/static/tftp
cd node_modules/on-tftp/static/tftp

for file in $(echo "\
monorail.ipxe \
monorail-undionly.kpxe \
monorail-efi64-snponly.efi \
monorail-efi32-snponly.efi");do
    wget "https://dl.bintray.com/rackhd/binary/ipxe/$file"
done

cd -
```

```
mkdir -p node_modules/on-http/static/http/common
cd node_modules/on-http/static/http/common

for file in $(echo "\
discovery.docker.tar.xz \
initrd-1.2.0-rancher \
vmlinuz-1.2.0-rancher");do
wget "https://dl.bintray.com/rackhd/binary/builds/$file"
done

cd -
```

3. Start RackHD

Start the 5 services of RackHD with pm2 and a yml file.

(a) Install pm2

```
sudo npm install pm2 -g
```

(a) Prepare a yml file

An example of yml file:

```
apps:
  - script: index.js
    name: on-taskgraph
    cwd: node_modules/on-taskgraph
  - script: index.js
    name: on-http
    cwd: node_modules/on-http
  - script: index.js
    name: on-dhcp-proxy
    cwd: node_modules/on-dhcp-proxy
  - script: index.js
    name: on-syslog
    cwd: node_modules/on-syslog
  - script: index.js
    name: on-tftp
    cwd: node_modules/on-tftp
```

(b) Start Services

```
sudo pm2 start rackhd.yml
```

All the services are started:

App name	id	mode	pid	status	restart	uptime	cpu	mem	watchin
on-dhcp-proxy	2	fork	16189	online	0	0s	60%	21.2 MB	disable
on-http	1	fork	16183	online	0	0s	100%	21.3 MB	disable
on-syslog	3	fork	16195	online	0	0s	60%	20.5 MB	disable
on-taskgraph	0	fork	16177	online	0	0s	6%	21.3 MB	disable
on-tftp	4	fork	16201	online	0	0s	66%	19.5 MB	disable

How to Erase the Database to Restart Everything

```
sudo pm2 stop rackhd.yml

mongo pxe
  db.dropDatabase()
  ^D

sudo pm2 start rackhd.yml
```

CentOS 7

Prerequisites NICs

1. Start with an centos 7 instance with 2 nics:
 - eno16777984 for the 'public' network - providing access to RackHD APIs, and providing routed (layer3) access to out of band network for machines under management
 - eno33557248 for dhcp/pxe to boot/configure the machines
2. Edit the network:
 - eno16777984 - assign IP address as appropriate for the environment, or you can use DHCP
 - eno33557248 static (172.31.128.0/22)
this is the 'default'. it can be changed, but more than one file needs to be changed.)

Packages

- **NodeJS 4.x**

1. **Remove Node.js (< 4.0)**

If Node.js is installed via yum, but is older than version 4.x, do this first .. code:

```
sudo yum remove nodejs
```

2. **Install Node.js 4.x**

Instructions copied from <https://github.com/nodesource/distributions#manual-installation>:

```
curl -sL https://rpm.nodesource.com/setup_4.x |sudo bash -
sudo yum install -y nodejs
```

Optional: install build tools

To compile and install native addons from npm you may also need to install build tools:

```
yum install gcc-c++ make
# or: yum groupinstall 'Development Tools'
```

3. **Verify Node.js 4.x**

```
$ node -v
v4.4.5
```

- **RabbitMQ**

1. Install Erlang

```
sudo yum -y update
sudo yum install -y epel-release
sudo yum install -y gcc gcc-c++ glibc-devel make ncurses-devel openssl-devel autoconf java

wget http://packages.erlang-solutions.com/erlang-solutions-1.0-1.noarch.rpm
sudo rpm -Uvh erlang-solutions-1.0-1.noarch.rpm
sudo yum -y update
```

2. Verify Erlang

```
erl
```

Sample output:

```
Erlang/OTP 19 [erts-8.2] [source-fbd2db2] [64-bit] [smp:8:8] [async-threads:10] [hipe] [

Eshell V8.2 (abort with ^G)
1>
```

3. Install RabbitMQ

```
wget https://www.rabbitmq.com/releases/rabbitmq-server/v3.6.1/rabbitmq-server-3.6.1-1.noarch.rpm
sudo rpm --import https://www.rabbitmq.com/rabbitmq-signing-key-public.asc
sudo yum install -y rabbitmq-server-3.6.1-1.noarch.rpm
```

4. Start RabbitMQ

```
sudo systemctl start rabbitmq-server
sudo systemctl status rabbitmq-server
```

• MongoDB

1. Configure the package management system (yum)

Create a /etc/yum.repos.d/mongodb-org-3.4.repo and add below lines:

```
[mongodb-org-3.4]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-org/3.4/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-3.4.asc
```

2. Install MongoDB

```
sudo yum install -y mongodb-org
```

3. Start MongoDB

```
sudo systemctl start mongod.service
sudo systemctl status mongod.service
```

• snmp

1. Install snmp

```
sudo yum install -y net-snmp
```

2. Start snmp

```
sudo systemctl start snmpd.service
sudo systemctl status snmpd.service
```

- **ipmitool**

```
sudo yum install -y OpenIPMI ipmitool
```

- **git**

1. Install git

```
sudo yum install -y git
```

2. Verify git

```
git --version
```

- **ansible**

1. Install ansible

```
sudo yum install -y ansible
```

2. Verify ansible

```
ansible --version
```

Sample output:

```
ansible 2.2.0.0
  config file = /etc/ansible/ansible.cfg
  configured module search path = Default w/o overrides
```

- **amtterm**

```
sudo yum install amtterm
```

- **dhcp**

```
sudo yum install -y dhcp
sudo cp /usr/share/doc/dhcp-4.2.5/dhcpd.conf.example /etc/dhcp/dhcpd.conf
```

Install & Configure RackHD

1. Install RackHD NPM Packages

Install the latest release of RackHD

```
for service in $(echo "on-dhcp-proxy on-http on-tftp on-syslog on-taskgraph");
do
  npm install $service;
done
```

2. Basic RackHD Configuration

- **DHCP**

Update /etc/dhcp/dhcpd.conf per your network configuration

```
# RackHD added lines
deny duplicates;

ignore-client-uids true;

subnet 172.31.128.0 netmask 255.255.240.0 {
    range 172.31.128.2 172.31.143.254;
    # Use this option to signal to the PXE client that we are doing proxy DHCP
    option vendor-class-identifier "PXEClient";
}
```

- **Open Ports in Firewall**

If the firewall is enabled, open below ports in firewall:

- 4011/udp
- 8080/tcp
- 67/udp
- 8443/tcp
- 69/udp
- 9080/tcp

An example of opening port:

```
sudo firewall-cmd --permanent --add-port=8080/tcp
sudo firewall-cmd --reload
```

- **CONFIGURATION FILE**

Create the required file /opt/monorail/config.json , you can use the demonstration configuration file at <https://github.com/RackHD/RackHD/blob/master/packer/ansible/roles/monorail/files/config.json> as a reference.

- **RACKHD BINARY SUPPORT FILES**

Download binary files from bintray and placed them with below shell script.

```
#!/bin/bash

mkdir -p node_modules/on-tftp/static/tftp
cd node_modules/on-tftp/static/tftp

for file in $(echo "\
monorail.ipxe \
monorail-undionly.kpxe \
monorail-efi64-snponly.efi \
monorail-efi32-snponly.efi");do
wget "https://dl.bintray.com/rackhd/binary/ipxe/$file"
done

cd -

mkdir -p node_modules/on-http/static/http/common
cd node_modules/on-http/static/http/common
```

```
for file in $(echo "\
discovery.docker.tar.xz \
initrd-1.2.0-rancher \
vmlinuz-1.2.0-rancher");do
wget "https://dl.bintray.com/rackhd/binary/builds/$file"
done

cd -
```

3. Start RackHD

Start the 5 services of RackHD with pm2 and a yml file.

(a) Install pm2

```
sudo npm install pm2 -g
```

(a) Prepare a yml file

An example of yml file:

```
apps:
- script: index.js
  name: on-taskgraph
  cwd: node_modules/on-taskgraph
- script: index.js
  name: on-http
  cwd: node_modules/on-http
- script: index.js
  name: on-dhcp-proxy
  cwd: node_modules/on-dhcp-proxy
- script: index.js
  name: on-syslog
  cwd: node_modules/on-syslog
- script: index.js
  name: on-tftp
  cwd: node_modules/on-tftp
```

(b) Start Services

```
sudo pm2 start rackhd.yml
```

All the services are started:

App name	id	mode	pid	status	restart	uptime	cpu	mem	watching
on-dhcp-proxy	2	fork	16189	online	0	0s	60%	21.2 MB	disabled
on-http	1	fork	16183	online	0	0s	100%	21.3 MB	disabled
on-syslog	3	fork	16195	online	0	0s	60%	20.5 MB	disabled
on-taskgraph	0	fork	16177	online	0	0s	6%	21.3 MB	disabled
on-tftp	4	fork	16201	online	0	0s	66%	19.5 MB	disabled

How to Erase the Database to Restart Everything

```
sudo pm2 stop rackhd.yml
```

```
mongo pxe
  db.dropDatabase()
^D

sudo pm2 start rackhd.yml
```

Configuration

The following JSON is an examples of the current defaults:

config.json

```
{
  "amqp": "amqp://localhost",
  "rackhdPublicIp": null,
  "apiServerAddress": "172.31.128.1",
  "apiServerPort": 9030,
  "dhcpPollerActive": false,
  "dhcpGateway": "172.31.128.1",
  "dhcpProxyBindAddress": "172.31.128.1",
  "dhcpProxyBindPort": 4011,
  "dhcpSubnetMask": "255.255.240.0",
  "gatewayaddr": "172.31.128.1",
  "trustedProxy": false,
  "httpEndpoints": [
    {
      "address": "0.0.0.0",
      "port": 8080,
      "httpsEnabled": false,
      "proxiesEnabled": true,
      "authEnabled": false,
      "yamlName": ["monorail-2.0.yaml", "redfish.yaml"]
    },
  ],
  "taskGraphEndpoint": {
    "address": "172.31.128.1",
    "port": 9030
  },
  "httpDocsRoot": "./build/apidoc",
  "httpFileServiceRoot": "./static/files",
  "httpFileServiceType": "FileSystem",
  "fileServerAddress": "172.31.128.2",
  "fileServerPort": 3000,
  "fileServerPath": "/",
  "httpProxies": [
    {
      "localPath": "/coreos",
      "server": "http://stable.release.core-os.net",
      "remotePath": "/amd64-usr/current/"
    }
  ],
  "httpStaticRoot": "/opt/monorail/static/http",
  "authTokenSecret": "RackHDRocks!",
  "authTokenExpireIn": 86400,
  "mongo": "mongodb://localhost/pxe",
  "sharedKey": "qxfo2D3tIJsZACu7UA6Fbw0avowo8r79ALzn+WeuC8M=",
  "statsd": "127.0.0.1:8125",
```



```

"syslogBindAddress": "172.31.128.1",
"syslogBindPort": 514,
"tftpBindAddress": "172.31.128.1",
"tftpBindPort": 69,
"tftpRoot": "./static/tftp",
"minLogLevel": 2,
"logColorEnable": false,
"enableUPnP": true,
"ssdpBindAddress": "0.0.0.0",
"heartbeatIntervalSec": 10,
"wssBindAddress": "0.0.0.0",
"wssBindPort": 9100
}

```

Configuration Parameters

The following table describes the configuration parameters in config.json:

Parameter	Description
amqp	<p>URI for accessing the AMQP interprocess communications channel. RackHD can be configured to use a single AMQP server or a AMQP cluster consisting of multiple AMQP servers.</p> <p>For a single AMQP server use the following formats:</p> <pre>"amqp": "amqp[s]://localhost",</pre> <pre>"amqp": "amqp[s]://<host>:<port>",</pre> <p>For multiple AMQP servers use an array with the following format:</p> <pre>"amqp": ["amqp[s]://<host_1>:<port_1>", "amqp[s]://</pre>
amqpSsl	<p>SSL setting used to access the AMQP channel. To enable SSL connections to the AMQP channel:</p> <pre>{ "enabled": true, "keyFile": "/path/to/key/file", "certFile": "/path/to/cert/file", "caFile": "/path/to/cacert/file" }</pre> <p>The key, certificate, and certificate authority files must be in pem format. Alternatively, pfxFile can be used to read key and certificate from a single file.</p>
apiServerAddress	External facing IP address of the API server
rackhdPublicIp	RackHD's public IP
apiServerPort	External facing port of the API server
dhcpPollerActive	Set to true to enable the dhcp isc lease poller (defaults to false)
dhcpLeasesPath	Path to dhcpd.leases file.
dhcpGateway	Gateway IP for the network for DHCP
dhcpProxyBindAddress	IP for DHCP proxy server to bind (defaults to '0.0.0.0'). Note: DHCP binds to 0.0.0.0 to support broadcast request/response within Node.js.
dhcpProxyBindPort	Port for DHCP proxy server to bind (defaults to 4011).

Continued on next page

Table 1.1 – continued from previous page

Parameter	Description
dhcpProxyOutPort	Port for DHCP proxy server to respond to legacy boot clients (defaults to 68).
dhcpProxyEFIOutPort	Port for DHCP proxy server to respond to EFI clients (defaults to 4011).
httpApiDocsDirectory	Fully-qualified directory containing the API docs.
httpEndpoints	Collection of http/https endpoints. See details in Setup HTTP/HTTPS endpoint
httpFileServiceRoot	Directory path for for storing uploaded files on disk.
httpFileServiceType	Backend storage mechanism for file service. Currently only FileSystem is supported.
fileServerAddress	Optional. Node facing IP address of the static file server. See Static File Service Setup .
fileServerPort	Optional. Port of the static file server. See Static File Service Setup .
fileServerPath	Optional. Access path of the static file server. See Static File Service Setup .
httpProxies	Optional HTTP/HTTPS proxies list. There are 3 parameters for each proxy: “localPath”/“remotePath” are optional and defaults to “/”. A legal “localPath”/“remotePath” string must start with slash and ends without slash, like “/mirrors”. If “localPath” is assigned to an existing local path like “/api/current/nodes”, proxy won’t work. Instead the path will keep its original feature and function. “server” is a must, both http and https servers are supported. A legal “server” string must ends without slash like “http://centos.eecs.wsu.edu”. Instead “http://centos.eecs.wsu.edu/” is illegal. Example: { “server”: “http://centos.eecs.wsu.edu”, “localPath”: “/centos” } would map http requests to local directory /centos/ to http://centos.eecs.wsu.edu/ { “server”: “https://centos.eecs.wsu.edu”, “remotePath”: “/centos” } would map http requests to local directory / to https://centos.eecs.wsu.edu/centos/ Note: To ensure this feature works, the httpProxies need be separately enabled for specified HTTP/HTTPS endpoint. See details in Setup HTTP/HTTPS endpoint
httpFrontendDirectory	Fully-qualified directory to the web GUI content
httpStaticDirectory	Fully-qualified directory to where static HTTP content is served
maxTaskPayloadSize	Maximum payload size expected through TASK runner API callbacks from microkernel
mongo	URI for accessing MongoDB. To support Mongo Replica Set feature, URI format is, mongodb://[username:password@]host1[:port1][,host2[:port2],...,hostN[:portN]

Continued on next page

Table 1.1 – continued from previous page

Parameter	Description
migrate	<p>The <i>migrate</i> setting controls the auto-migration strategy that every time RackHD loads, the strategy should be one of <i>safe</i>, <i>alter</i> and <i>drop</i>.</p> <p>NOTE: It's extremely important to set the <i>mi-grate</i> to <i>safe</i> when working with existing databases, otherwise, you will very likely lose data! The <i>alter</i> and <i>drop</i> strategies are only recommended in development environment. You could see detail description for each migration strategy from this link https://github.com/balderdashy/sails-docs/blob/master/concepts/ORM/model-settings.md#migrate</p> <p>The RackHD default migration strategy is <i>safe</i>.</p>
sharedKey	<p>A 32 bit base64 key encoded string relevant for aes-256-cbc, defaults to 'qxfO2D3tIJszACu7UA6Fbw0avowo8r79ALzn+WeuC8M='.</p> <p>The default can be replaced by a 256 byte randomly generated base64 key encoded string.</p> <p>Example generating a key with OpenSSL:</p> <pre>openssl enc -aes-256-cbc -k secret -P -md sha1</pre>
obmInitialDelay	Delay before retrying an OBM invocation
obmRetries	Number of retries to attempt before failing an OBM invocation
pollerCacheSize	Maximum poller entries to cache in memory
statsdPrefix	Application-specific <i>statsd</i> metrics for debugging
syslogBindPort	Port for syslog (defaults to 514).
syslogBindAddress	Address for the syslog server to bind to (defaults to '0.0.0.0').
tftpBindAddress	Address for TFTP server to bind to (defaults to '0.0.0.0').
tftpBindPort	Listening port for TFTP server (defaults to 69).
tftpBindAddress	File root for TFTP server to serve files (defaults to './static/tftp').
tftpboot	Fully-qualified directory from which static TFTP content is served
minLogLevel	A numerical value for filtering the logging from RackHD. The log levels for filtering are defined at https://github.com/RackHD/on-core/blob/master/lib/common/constants.js#L31-L37
logColorEnable	A boolean value to toggle the colorful log output (defaults to false)
enableLocalHostException	Set to true to enable the localhost exception, see Setup the First User with Localhost Exception .
enableUPnP	Set to true to advertise RackHD Restful API services using SSDP (Simple Service Discovery Protocol).
ssdpBindAddress	The bind address to send the SSDP advertisements on (defaults to 0.0.0.0).

Continued on next page

Table 1.1 – continued from previous page

Parameter	Description
heartbeatIntervalSec	Integer value setting the heartbeat send interval in seconds. Setting this value to 0 will disable the heartbeat service (defaults to 10)
wssBindAddress	Address for RackHD WebSocket Service to bind to (defaults to '0.0.0.0').
wssBindPort	Listening port for RackHD WebSocket Service (defaults to 9100).
trustedProxy	Enable trust proxy in express. Populate req.ip with left most IP address from the XForwardFor list.
discoveryGraph	Injectable name for the discovery graph that should be run against new nodes See documentation at https://expressjs.com/en/guide/behind-proxies.html
autoCreateObm	Allow rackHD to setup IPMI OBM settings on active discovery by creating a new BMC user on the compute node.

These configurations can also be overridden by setting environment variables in the process that's running each application, or on the command line when running node directly. For example, to override the value of amqp for the configuration, you could use:

```
export amqp=amqp://another_host:5763
```

prior to running the relevant application.

HTTPS/TLS Configuration

To use TLS, a private RSA key and X.509 certificate must be provided. On Ubuntu and Mac OS X, the openssl command line tool can be used to generate keys and certificates.

For internal development purposes, a self-signed certificate can be used. When using a self-signed certificate, clients must manually include a rule to trust the certificate's authenticity.

By default, the application uses a self-signed certificate issued by Monorail which requires no configuration. Custom certificates can also be used with some configuration.

Parameters

See the table in [Configuration Parameters](#) for information about HTTP/HTTPS configuration parameters. These parameters begin with *HTTP* and *HTTPS*.

BMC Username and Password Configuration

A node gets discovered and the BMC IPMI comes up with a default username/password. User can automatically set IPMI OBM settings using a default user name('__rackhd__') and an auto generated password in rackHD by adding the following to RackHD config.json:

```
"autoCreateObm": "true"
```

If a user wants to change the BMC credentials later in time, when the node has been already discovered and database updated, a separate workflow located at `on-taskgraph/lib/graphs/bootstrap-bmc-credentials-setup-graph.js` can be posted using Postman or Curl command.

POST: <http://server-ip:8080/api/current/workflows/>

add the below content in the json body for payload (example node identifier and username, password shown below)

```
{
  "name": "Graph.Bootstrap.With.BMC.Credentials.Setup",
  "options": {
    "defaults": {
      "graphOptions": {
        "target": "56e967f5b7a4085407da7898",
        "generate-pass": {
          "user": "7",
          "password": "7"
        }
      },
      "nodeId": "56e967f5b7a4085407da7898"
    }
  }
}
```

By running this workflow, a boot-graph runs to bootstrap an ubuntu image on the node again and set-bmc-credentials-graph runs the required tasks to update the BMC credentials. Below is a snippet of the 'Bootstrap-And-Set-Credentials-graph', when the graph is posted the node reboots and starts the discovery process

```
module.exports = {
  friendlyName: 'Bootstrap And Set Credentials',
  injectableName: 'Graph.Bootstrap.With.BMC.Credentials.Setup',
  options: {
    defaults: {
      graphOptions: {
        target: null
      },
      nodeId: null
    }
  },
  tasks: [
    {
      label: 'boot-graph',
      taskDefinition: {
        friendlyName: 'Boot Graph',
        injectableName: 'Task.Graph.Run.Boot',
        implementsTask: 'Task.Base.Graph.Run',
        options: {
          graphName: 'Graph.BootstrapUbuntu',
          defaults: {
            graphOptions: { }
          }
        },
        properties: {}
      }
    },
    {
      label: 'set-bmc-credentials-graph',
      taskDefinition: {
        friendlyName: 'Run BMC Credential Graph',
        injectableName: 'Task.Graph.Run.Bmc',
        implementsTask: 'Task.Base.Graph.Run',
        options: {
          graphName: 'Graph.Set.Bmc.Credentials',

```

```
        defaults : {
            graphOptions: {    }
        },
        properties: {}
    },
    waitOn: {
        'boot-graph': 'finished'
    }
},
{
    label: 'finish-bootstrap-trigger',
    taskName: 'Task.Trigger.Send.Finish',
    waitOn: {
        'set-bmc-credentials-graph': 'finished'
    }
}
]
};
```

To remove the BMC credentials, User can run the following workflow located at `on-taskgraph/lib/graphs/bootstrap-bmc-credentials-remove-graph.js` and can be posted using Postman or Curl command.

POST: <http://server-ip:8080/api/current/workflows/>

add the below content in the json body for payload (example node identifier and username, password shown below)

```
{
  "name": "Graph.Bootstrap.With.BMC.Credentials.Remove",
  "options": {
    "defaults": {
      "graphOptions": {
        "target": "56e967f5b7a4085407da7898",
        "remove-bmc-credentials": {
          "users": ["7", "8"]
        }
      },
      "nodeId": "56e967f5b7a4085407da7898"
    }
  }
}
```

Certificates

This section describes how to generate and install a self-signed certificate to use for testing.

Generating Self-Signed Certificates

If you already have a key and certificate, skip down to the *Installing Certificates* section.

First, generate a new RSA key:

```
openssl genrsa -out privkey.pem 2048
```

The file is output to `privkey.pem`. **Keep this private key secret. If it is compromised, any corresponding certificate should be considered invalid.**

The next step is to generate a self-signed certificate using the private key:

```
openssl req -new -x509 -key privkey.pem -out cacert.pem -days 9999
```

The *days* value is the number of days until the certificate expires.

When you run this command, OpenSSL prompts you for some metadata to associate with the new certificate. The generated certificate contains the corresponding public key.

Installing Certificates

Once you have your private key and certificate, you'll need to let the application know where to find them. It is suggested that you move them into the `/opt/monorail/data` folder.

```
mv privkey.pem /opt/monorail/data/mykey.pem
mv cacert.pem /opt/monorail/data/mycert.pem
```

Then configure the paths by editing `httpsCert` and `httpKey` in `/opt/monorail/config.json`. (See the [Configuration Parameters](#) section above).

If using a self-signed certificate, add a security exception to your client of choice. Verify the certificate by restarting on-http and visiting `https://<host>/api/current/versions`.

Note: For information about OpenSSL, see the [OpenSSL documentation](#).

Setup HTTP/HTTPS endpoint

This section describes how to setup HTTP/HTTPS endpoints in RackHD. An endpoint is an instance of HTTP or HTTPS server that serves a group of APIs. Users can choose to enable authentication or enable HTTPS for each endpoint.

There is currently one API group defined in RackHD:

- the northbound-api-router API group. This is the API group that is used by users

```
[
  {
    "address": "0.0.0.0",
    "port": 8443,
    "httpsEnabled": true,
    "httpsCert": "data/dev-cert.pem",
    "httpsKey": "data/dev-key.pem",
    "httpsPfx": null,
    "proxiesEnabled": false,
    "authEnabled": false,
    "yamlName": ["monorail-2.0.yaml", "redfish.yaml"]
  }
]
```

Parameter	Description
address	IP/Interface to bind to for HTTP. Typically this is '0.0.0.0'
port	Local port to use for HTTP. Typically, port 80 for HTTP, 443 for HTTPS
httpsEnabled	Toggle HTTPS
httpsCert	Filename of the X.509 certificate to use for TLS. Expected format is PEM. This is optional and only takes effect when the httpsEnabled flag is set to true
httpsKey	Filename of the RSA private key to use for TLS. Expected format is PEM. This is optional and only takes effect when the httpsEnabled flag is set to true
httpsPfx	Pfx file containing the SSL cert and private key (only needed if the key and cert are omitted) This is optional and only takes effect when the httpsEnabled flag is set to true
proxiesEnabled	A boolean value to toggle httpProxies (defaults to false)
authEnabled	Toggle API Authentication
yaml-Name	A list of yaml file used to define the routes. current available files are momorail-2.0.yaml, and redfish.yaml.

Setup Taskgraph Endpoint

This section describes how to setup the taskgraph endpoint in RackHD. The taskgraph endpoint is the interface that is used by nodes to interacting with the system

```
"taskGraphEndpoint": {
  "address": "172.31.128.1",
  "port": 9030
}
```

Parameter	Description
address	IP/Interface that the taskgraph service is listening on
port	Local port that the taskgraph service is listening on

Raid Configuration

Setting up the docker image

For the correct tooling (storcli for Quanta/Intel and perccli for Dell) you will need to build the docker image using the following steps:

- (1). Add the repo <https://github.com/RackHD/on-imagebuilder>
- (2). Refer to the Requirements section of the Readme in the on-imagebuilder repo to install latest version of docker: <https://github.com/RackHD/on-imagebuilder#requirements>
- (3). For Quanta/Intel storcli - <https://github.com/RackHD/on-imagebuilder#oem-tools>

Refer to the **OEM tools** section: OEM docker images **raid** and **secure_erase** require storcli_1.17.08_all.deb being copied into raid and secure-erase under on-imagebuilder/oem. User can download it from http://docs.avagotech.com/docs/1.17.08_StorCLI.zip

- (4). For Dell PERCcli: <https://github.com/RackHD/on-imagebuilder#oem-tools>

Refer to the **OEM tools** section to download and unzip the percCLI package and derive a debian version using 'alien' There is no .deb version perccli tool. User can download .rpm perccli from

https://downloads.dell.com/FOLDER02444760M/1/perccli-1.11.03-1_Linux_A00.tar.gz unzip the package and then use alien to get a .deb version perccli tool as below:

```
sudo apt-get install alien
sudo alien -k perccli-1.11.03-1.noarch.rpm
```

OEM docker images **dell_raid** and **secure_erase** require perccli_1.11.03-1_all.deb being copied into dell-raid and secure-erase under on-imagebuilder/oem.

(5). Build the docker image.

```
#This creates the dell.raid.docker.tar.xz image
cd on-imagebuilder/oem/dell-raid
sudo docker build -t rackhd/micro .
sudo docker save rackhd/micro | xz -z > dell.raid.docker.tar.xz

#This creates the raid.docker.tar.xz image
cd on-imagebuilder/oem/raid
sudo docker build -t rackhd/micro .
sudo docker save rackhd/micro | xz -z > raid.docker.tar.xz
```

(6). Copy the image dell.raid.docker.tar.xz or raid.docker.tar.xz to /on-http/static/http/common

(7). Restart the RackHD service

Posting the Workflow

POST: <http://server-ip:8080/api/2.0/nodes/:id/workflows/?name=Graph.Bootstrap.Megaraid.Configure>

add the below example content in the json body for payload

```
{
  "options": {
    "config-raid": {
      "ssdStoragePoolArr": [],
      "ssdCacheCadeArr": [{
        "enclosure": 252,
        "type": "raid0",
        "drives": "[0]"
      }],
      "controller": 0,
      "path": "/opt/MegaRAID/storcli/storcli64",
      "hddArr": [{
        "enclosure": 252,
        "type": "raid0",
        "drives": "[1]"
      },
      {
        "enclosure": 252,
        "type": "raid1",
        "drives": "[4,5]"
      }
    ]
  }
}
```

Notes: ssdStoragePoolArr, ssdCacheCadeArr, hddArr should be passed as empty arrays if they don't need to be configure like the "ssdStoragePoolArr" array in the example payload above is an empty array. For CacheCade (ssd-CacheCadeArr) to work the controller should have the ability to configure it.

Payload Definition

The drive information for payload can be gathered from the node catalogs using the api below:

```
GET /api/current/nodes/<id>/catalogs/<source>
```

Or from the node's microkernel: (Note: the workflow does not stop in the micro-kernel. In order to be able to stop in the microkernel the workflow needs to be updated to remove the last two tasks.)

```
{
  label: 'refresh-catalog-megaraid',
  taskName: 'Task.Catalog.megaraid',
  waitOn: {
    'config-raid': 'succeeded'
  }
},
{
  label: 'final-reboot',
  taskName: 'Task.Obm.Node.Reboot',
  waitOn: {
    'refresh-catalog-megaraid': 'finished'
  }
}
```

The elements in the arrays represent the EID of the drives (run this command in the micro-kernel storcli 64 /c0 show)

```
Physical Drives = 6 PD LIST : =====
EID:SlT DID State DG Size Intf Med SED PI SeSz Model Sp
-----
252:0 0 Onln 0 372.093 GB SAS SSD N N 512B HUSMM1640ASS200 U
252:1 4 Onln 5 1.090 TB SAS HDD N N 512B HUC101212CSS600 U
252:2 3 Onln 1 1.090 TB SAS HDD N N 512B HUC101212CSS600 U
252:4 5 Onln 2 1.090 TB SAS HDD N N 512B HUC101212CSS600 U
252:5 2 Onln 3 1.090 TB SAS HDD N N 512B HUC101212CSS600 U
252:6 1 Onln 4 1.090 TB SAS HDD N N 512B HUC101212CSS600 U
```

“hddArr”: is the array of hard drives that will take part of the storage pool “ssdStoragePoolArr”: is the array of solid state drives that will take part of the storage pool “ssdCacheCadeArr”: is the array of hard drives that will take part of CacheCade

Results

After the workflow runs successfully, you should be able to see the newly created virtual disks either from the catalogs or from the monorail micro-kernel

```
monorail@monorail-micro:~$ sudo /opt/MegaRAID/storcli/storcli64 /c0/vall show Virtual Drives : =====
0/0 Cac0 Opt1 RW Yes NRWBD - ON 372.093 GB
1/1 RAID0 Opt1 RW Yes RWTD - ON 1.090 TB
2/2 RAID0 Opt1 RW Yes RWTD - ON 1.090 TB
3/3 RAID0 Opt1 RW Yes RWTD - ON 1.090 TB
4/4 RAID0 Opt1 RW Yes RWTD - ON 1.090 TB
5/5 RAID0 Opt1 RW Yes RWTD - ON 1.090 TB
```

Security

Authentication

When 'authEnabled' is set to 'true' in the config.json file for an endpoint, authentication will be needed to access the APIs that are defined within that endpoint. Enabling authentication will also enable authorization control when accessing API 2.0 and Redfish APIs.

This section describes how to access APIs that need authentication.

Enable Authentication

Please refer to [Setup HTTP/HTTPS endpoint](#) on how to setup endpoints. Simply put, the following endpoint configuration will be a good start.

```
"httpEndpoints": [
  {
    "address": "0.0.0.0",
    "port": 8443,
    "httpsEnabled": true,
    "proxiesEnabled": false,
    "authEnabled": true,
    "routers": "northbound-api-router"
  },
  {
    "address": "172.31.128.1",
    "port": 8080,
    "httpsEnabled": false,
    "proxiesEnabled": false,
    "authEnabled": false,
    "routers": "southbound-api-router"
  }
]
```

The first endpoint represents an HTTPS service listening at port 8443 that serves northbound APIs, which are APIs being called by users. Note that authEnabled is set to true means that authentication is needed to access northbound APIs.

The second endpoint represents an HTTP service listening at port 8080 that serves southbound APIs, which are called by nodes interacting with the system. Authentication should NOT be enabled for southbound APIs in order for PXE to work fine.

Note: although there is no limitation to enable authentication together with insecure HTTP (httpsEnabled = false) for an endpoint, it is strongly not recommended to do so. Sending user credentials over unencrypted HTTP connection exposes users to the risk of malicious attacks.

Setup the First User with Localhost Exception

The localhost exception permits unauthenticated access to create the first user in the system. With authentication enabled, the first user can be created by issuing a POST to the /users API only if the API is issued from localhost. The first user must be assigned a role with privileges to create other users, such as an Administrator role.

Here is an example of creating an initial 'admin' user with a password of 'admin123'.

```
curl -ks -X POST -H "Content-Type:application/json" https://localhost:8443/api/current/users -d '{"us
{
```

```
"role": "Administrator",
"username": "admin"
}
```

The localhost exception can be disabled by setting the configuration value “enableLocalHostException” to false. The default value of “enableLocalHostException” is true.

Setup the Token

There are few settings needed for generating the token.

Parameter	Description
authTo- kenSecret	The secret used to generate the token.
authToken- ExpireIn	The time interval in second after which the token will expire, since the time the token is generated. Token will never expire if this value is set to 0.

Login to Get a Token

Following the endpoint settings, a token is needed to access any northbound APIs, except the /login API.

Posting a request to /login with username and password in the request body will get a token returned from RackHD, which will be used to access any other northbound APIs.

Here is an example of getting a token using curl.

```
curl -k -X POST -H "Content-Type:application/json" https://localhost:8443/login -d '{"username":"admin", "password":"1234567890"}'
```

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current
			Dload	Upload	Total	Spent	Left
100	204	100	160	100	44	3315	911
--:--:--	--:--:--	--:--:--	--:--:--	--:--:--	--:--:--	--:--:--	3333
{							
"token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoiYWRtaW4iLCJpYXQoIjE0NTU2MTI5MzMsImV4c2V5IjoiZm9udCJ9.eyJ1c2VyIjoiYWRtaW4iLCJpYXQoIjE0NTU2MTI5MzMsImV4c2V5IjoiZm9udCJ9"							
}							

A 401 unauthorized response with 'Invalid username or password' message will be returned if:

- Username or password is wrong in the http request body

For example:

```
curl -k -X POST -H "Content-Type:application/json" https://localhost:8443/login -d '{"username":"admin", "password":"1234567890"}'
```

% Total	% Received	% Xferd	Average Speed		Time	Time	Time	Current
			Dload	Upload	Total	Spent	Left	Speed
100	94	100	42	100	52	909	1125	--:--:-- --:--:-- --:--:-- 1130
<pre>{ "message": "Invalid username or password" }</pre>								

Accessing API Using the Token

There are three ways of using the token in a http/https request:

- send the token as a query string
- send the token as a query header
- send the token as request body

- No authorization key in request header

For example:

```
curl -k -H "Content-Type:application/json" https://localhost:8443/api/1.1/config | python -mjson.tool
      Dload   Upload   Total   Spent    Left   Speed
100    27   100    27     0     0   1644     0 --:--:-- --:--:-- --:--:--   1687
{
  "message": "No auth token"
}
```

Invalidating all Tokens

All active tokens can be invalidated by changing the `authTokenSecret` property in the RackHD configuration file:

`config.json`

Edit `config.json`, modify the value of `authTokenSecret`, and save the file. Restart the `on-http` service. Any previously generated tokens, signed with the old secret, will now be invalid.

Creating a Redfish Session

Posting a request to the Redfish Session Service with `UserName` and `Password` in the request body will get a token returned from the Redfish service which can be used to access any other Redfish APIs. The token is returned in the 'X-Auth-Token' header in the response object.

Here is an example of getting a token using `curl`.

```
curl -vk -X POST -H "Content-Type:application/json" https://localhost:8443/redfish/v1/SessionService/
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Access-Control-Allow-Origin: *
< X-Auth-Token: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2V2YiIjoIYWRTaW4iLCJpZCI6ImNlYjk0MzIzLTQyZDYy
< Content-Type: application/json; charset=utf-8
< Content-Length: 294
< ETag: W/"126-K9SNCTT10D9033EnNBAPcQ"
< Date: Mon, 12 Sep 2016 19:09:58 GMT
< Connection: keep-alive
<
{ [data not shown]
100   338  100   294  100    44   4785    716 --:--:-- --:--:-- --:--:--   4819
* Connection #0 to host localhost left intact
{
  "@odata.context": "/redfish/v1/$metadata#SessionService/Sessions/Members/$entity",
  "@odata.id": "/redfish/v1/SessionService/Sessions",
  "@odata.type": "#Session.1.0.0.Session",
  "Description": "User Session",
  "Id": "ceb94323-42d6-4c70-9d21-105f2a8e5cc8",
  "Name": "User Session",
  "Oem": {},
  "UserName": "admin"
}
```

A 401 unauthorized response will be returned if:

- Username or password is wrong in the http request body

For example:


```
< Date: Mon, 12 Sep 2016 20:04:32 GMT
< Connection: keep-alive
<
{ [data not shown]
100      2 100      2    0    0    64      0 --:--:-- --:--:-- --:--:--      66
* Connection #0 to host localhost left intact
{}
```

Authorization

API access control is enabled when authentication is enabled. The Access Control is controlled per API and per API method. A GET on an API can have different access control than a POST on the same API.

Privileges

A privilege grants access to an API resource and an action to perform on that resource. For example, a 'read' privilege may grant GET access on a set of APIs, but may not also grant POST/PUT/PATCH/DELETE access to those same APIs. To issue POST/PUT/PATCH/DELETE methods to an API, a 'write' privilege may be required.

Built-in Privileges The following Privileges are built-in to RackHD:

Privilege	Description
Read	Used to specify an ability to read data from an API
Write	Used to specify an ability to write data to an API
Login	Used to specify an ability to login to RackHD
ConfigureUsers	Used to specify an ability to configure aspects of other users
ConfigureSelf	Used to specify an ability to configure aspects of the logged in user
ConfigureManager	Used to specify an ability to configure Manager resources
ConfigureComponents	Used to specify an ability to configure components managed by this service

Roles

A role grants a set of privileges. Each privilege is specified explicitly within the role. Authenticated users have a single role assigned to them.

Built-in Roles The following Roles are built-in to RackHD:

Role	Description
Administrator	Possess all built-in privileges
ReadOnly	Possess Read, Login and ConfigureSelf privileges
Operator	Possess Login, ConfigureComponents, and ConfigureSelf privileges

API Commands for Roles The following API commands can be used to view, create, modify and delete roles.

Get a list of all roles currently stored in the system

```
GET /api/current/roles
```

Get information about a specified role.


```
GET /api/current/roles/<name>
```

Create a new role and store it.

```
POST /api/current/roles

{
  "privileges": [
    <privilege1>,
    <privilege2>
  ]
  "role": "<name>"
}
```

Modify the properties of a specified role.

```
PATCH /api/current/roles/<name>

{
  "privileges": [
    <privilege1>,
    <privilege2>
  ]
}
```

Delete a specified role.

```
DELETE /api/current/roles/<name>
```

RackHD API

Our REST based API is the abstraction layer for the low-level management tasks that are performed on hardware devices, and information about those devices. For example, when a compute server is “discovered” (see [System Architecture](#) for more details on this process), the information about that server is expressed as *nodes* and *catalogs* in the RackHD API. When you want to re-image that compute node, the RackHD API is used to activate a workflow containing the tasks that are appropriate to doing that function.

The RackHD API can be used to manage nodes, catalogs, workflows, tasks, templates, pollers, and other entities. For the complete list of functions, generate the RackHD API documentation as described below or download the latest from <https://bintray.com/rackhd/docs/apidoc#files>.

List All Nodes

```
curl http://<server>:8080/api/current/nodes | python -mjson.tool
```

Get the Active Workflow

```
curl http://<server>:8080/api/current/nodes/<identifier>/workflows/?active=true | python -mjson.tool
```

Starting and Stopping the API Server

The API server runs by default. Use the following commands to stop or start the API server.

Action	Command
Stop API server	<i>sudo service on-http stop</i>
Start API server	<i>sudo service on-http start</i>

Generating API Documentation

You can generate an HTML version of the API documentation by cloning the *on-http* repository and running the following command.

```
$ git clone https://github.com/RackHD/on-http
$ cd on-http
$ npm install
$ npm run apidoc
$ npm run taskdoc
```

The default and example quick start build that we describe in `../tutorials/vagrant` has the API docs rendered and embedded within that instance for easy use, available at `http://[IP ADDRESS OF VM]:8080/docs/` for the 1.1 API documentation, and `http://[IP ADDRESS OF VM]:8080/swagger-ui/` for the current (2.0) and Redfish API documentation.

RackHD Client Libraries

The 2.0 API generates a swagger API definition file that can be used to create client libraries with *swagger*. To create this file locally, you can check out the *on-http* library and run the commands:

```
npm install
npm run apidoc
```

The resulting files will be in `build/swagger-doc` and will be pdf files that are documentation for the 2.0 API (`rackhd-api-2.1.0.pdf`) and the Redfish API (`rackhd-redfish-v1-1.1.1.pdf`).

To create a client library you can run the command:

```
npm run client -- -l <language>
```

Where the *language* you input can currently be `python`, `go`, or `java`. `Go` is generated using `go-swagger` and `python` and `java` are generated using `swagger-codegen`. This command will generate client libraries for the 2.0 API and Redfish API and will be saved in the directories `on-http/on-http-api2.0` and `on-http/on-http-redfish-1.0`, respectively.

You can also use the *swagger generator* online tool to generate a client zip bundle for a variety of languages, including `python`, `Java`, `javascript`, `ruby`, `scala`, `php`, and more.

Examples using the python client library

Getting a list of nodes

```
from on_http import NodesApi, ApiClient, Configuration

config = Configuration()
config.debug = True
config.verify_ssl = False

client = ApiClient(host='http://localhost:9090', header_name='Content-Type', header_value='application/json')
nodes = NodesApi(api_client=client)
nodes.api2_0_nodes_get()
print client.last_response.data
```

Deprecated 1.1 API - Getting a list of nodes:

```

from on_http import NodesApi, ApiClient, Configuration

config = Configuration()
config.debug = True
config.verify_ssl = False

client = ApiClient(host='http://localhost:9090', header_name='Content-Type', header_value='application,
nodes = NodesApi(api_client=client)
nodes.api1_1_nodes_get()
print client.last_response.data

```

Or the same asynchronously (with a callback):

```

def cb_func(resp):
    print 'GET /nodes callback!', resp

thread = nodes.api2_0_nodes_get(callback=cb_func)

```

Deprecated 1.1 API - Or the same asynchronously (with a callback):

```

def cb_func(resp):
    print 'GET /nodes callback!', resp

thread = nodes.api1_1_nodes_get(callback=cb_func)

```

Using Pagination

The RackHD 2.0 /nodes, /pollers, and /workflows APIs support pagination using \$skip and \$top query parameters.

Parameter	Description
\$skip	An integer indicating the number of items that should be skipped starting with the first item in the collection.
\$top	An integer indicating the number of items that should be included in the response.

These parameters can be used individually or combined to display any subset of consecutive resources in the collection.

Here is an example request using \$skip and \$top to get the second page of nodes with four items per page.

```
curl http://localhost:8080/api/current/nodes?$skip=4&$top=4
```

RackHD will add a link header to assist in traversing a large collection. Links will be added if either \$skip or \$top is used and the size of the collection is greater than the number of resources displayed (i.e. the collection cannot fit on one page). If applicable, links to first, last, next, and previous pages will be included in the header. The next and previous links will be omitted for the last and first pages respectively.

Here is an example link header from a collection containing 1000 nodes.

```

</api/current/nodes?$skip=0&$top=4>; rel="first",
</api/current/nodes?$skip=1004&$top=4>; rel="last",
</api/current/nodes?$skip=0&$top=4>; rel="prev",
</api/current/nodes?$skip=8&$top=4>; rel="next"

```

Event Notification

RackHD supports event notification via both web hook and AMQP.

A web hook allows applications to subscribe certain RackHD published events by configured URL, when one of the subscribed events is triggered, RackHD will send a POST request with event payload to configured URL.

RackHD also publishes defined events over AMQP, so subscribers to RackHD's instance of AMQP don't need to register a webhook URL to get events. The AMQP events can be prolific, so we recommend that consumers filter events as they are received to what is desired.

Events Payloads

All published external events' payload formats are common, the event attributes are as below:

At-tribute	Type	Description
version	String	Event payload format version.
type	String	It could be one of the values: heartbeat , node , polleralert , graph .
action	String	a verb or a composition of component and verb which indicates what happened, it's associated with the <i>type</i> attribute.
severity	String	Event severity, it could be one of the values: critical , warning , information .
typeId	String	It's associated with the <i>type</i> attribute. It could be graph 'Id' for graph type, poller 'Id' for polleralert type, <fqdn>.<service name> for heartbeat event, node 'Id' for node type. Please see table for more details .
createdAt	String	The time event happened.
nodeId	String	The node <i>Id</i> , it's <i>null</i> for 'heartbeat' event.
data	Object	Detail information are included in this attribute.

The table of *type*, *typeId*, *action* and *severity* for all external events

Example of heartbeat event payload:

```
{
  "version": "1.0",
  "type": "heartbeat",
  "action": "updated",
  "typeId": "kickseed.example.com.on-taskgraph",
  "severity": "information",
  "createdAt": "2016-07-13T14:23:45.627Z",
  "nodeId": "null",
  "data": {
    "name": "on-taskgraph",
    "title": "node",
    "pid": 6086,
    "uid": 0,
    "platform": "linux",
    "release": {
      "name": "node",
      "lts": "Argon",
      "sourceUrl": "https://nodejs.org/download/release/v4.7.2/node-v4.7.2.tar.gz",
      "headersUrl": "https://nodejs.org/download/release/v4.7.2/node-v4.7.2-headers.tar.gz"
    },
    "versions": {
      "http_parser": "2.7.0",
      "node": "4.7.2",
```

```

    "v8": "4.5.103.43",
    "uv": "1.9.1",
    "zlib": "1.2.8",
    "ares": "1.10.1-DEV",
    "icu": "56.1",
    "modules": "46",
    "openssl": "1.0.2j"
  },
  "memoryUsage": {
    "rss": 116531200,
    "heapTotal": 84715104,
    "heapUsed": 81638904
  },
  "currentTime": "2017-01-24T07:18:49.236Z",
  "nextUpdate": "2017-01-24T07:18:59.236Z",
  "lastUpdate": "2017-01-24T07:18:39.236Z",
  "cpuUsage": "NA"
}

```

Example of node *discovered* event payload:

```

{
  "type": "node",
  "action": "discovered",
  "typeId": "58aa8e54ef2b49ed6a6cdd4c",
  "nodeId": "58aa8e54ef2b49ed6a6cdd4c",
  "severity": "information",
  "data": {
    "ipMacAddresses": [
      {
        "ipAddress": "172.31.128.2",
        "macAddress": "2c:60:0c:ad:d5:ba"
      },
      {
        "macAddress": "90:e2:ba:91:1b:e4"
      },
      {
        "macAddress": "90:e2:ba:91:1b:e5"
      },
      {
        "macAddress": "2c:60:0c:c0:a8:ce"
      }
    ],
    "nodeId": "58aa8e54ef2b49ed6a6cdd4c",
    "nodeType": "compute"
  },
  "version": "1.0",
  "createdAt": "2017-02-20T06:37:23.775Z"
}

```

Events via AMQP

AMQP Exchange and Routing Key

The change of resources managed by RackHD could be retrieved from AMQP messages.

- Exchange: **on.events**
- Routing Key **<type>.<action>.<severity>.<typeId>.<nodeId>**

All the fields in routing key exists in the common event payloads *event_payload*.

Examples of routing key:

Heartbeat event routing key of on-tftp service:

```
heartbeat.updated.information.kickseed.example.com.on-tftp
```

Polleralert sel event routing key:

```
polleralert.sel.updated.critical.44b15c51450be454180fabcd.57b15c51450be454180fa460
```

Node discovered event routing key:

```
node.discovered.information.57b15c51450be454180fa460.57b15c51450be454180fa460
```

Graph event routing key:

```
graph.started.information.35b15c51450be454180fabd.57b15c51450be454180fa460
```

AMQP Routing Key Filter

All the events could be filtered by routing keys, for example:

All services' heartbeat events:

```
$ sudo node sniff.js "on.events" "heartbeat.#"
```

All nodes' discovered events:

```
$ sudo node sniff.js "on.events" "#.discovered.#"
```

'sniff.js' is a tool located at https://github.com/RackHD/on-tools/blob/master/dev_tools/README.md

Events via Hook

Register Web Hooks

The web hooks used for subscribing event notification could be registered by POST `<server>/api/current/hooks` API as below

```
curl -H "Content-Type: application/json" -X POST -d @payload.json <server>api/current/hooks
```

The *payload.json* attributes in the example above are as below:

At-tribute	Type	Flags	Description
url	String	re-quired	The hook url that events are notified to. Both http and https urls are supported. url must be unique.
name	String	op-tional	Any name user specified for the hook.
filters	Ar-ray	op-tional	An array of conditions that decides which events should be notified to hook url.

When a hook is registered and eligible events happened, RackHD will send a `POST` request to the hook url. POST request's Content-Type will be `application/json`, and the request body be the event payload.

An example of *payload.json* with minimal attributes:

```
{
  "url": "http://www.abc.com/def"
}
```

When multiple hooks are registered, a single event can be sent to multiple hook urls if it meets hooks' filtering conditions.

Event Filter Rules

The conditions of which events should be notified could be specified in the *filters* attribute in the *hook_payload*, when *filters* attribute is not specified, or it's empty, all the events will be notified to the hook url.

The *filters* attribute is an array, so multiple filters could be specified. The event will be sent as long as any filter condition is satisfied, even if the conditions may have overlaps.

The filter attributes are *type*, *typeId*, *action*, *severity* and *nodeId* listed in *event_payload*. Filtering by *data* is not supported currently. Filtering expression of hook *filters* is based on javascript regular expression, below table describes some base operations for hook filters:

Description	Example	Eligible Events
Attribute equals some value	{ "action": "^discovered\$" }	Events with <i>action</i> equals <i>discovered</i>
Attribute can be any of specified value.	{ "action": "discovered updated" }	Events with <i>action</i> equals either <i>discovered</i> or <i>updated</i>
Attribute can not be any of specified value.	{ "action": "[^(discovered updated)]" }	Events with <i>action</i> equals neither <i>discovered</i> nor <i>updated</i>
Multiple attributes must meet specified values.	{ "action": "[^(discovered updated)]", "type": "node" }	Events with <i>type</i> equals <i>node</i> while <i>action</i> equals neither <i>discovered</i> nor <i>updated</i>

An example of multiple filters:

```
{
  "name": "event sets",
  "url": "http://www.abc.com/def",
  "filters": [
    {
      "type": "node",
      "nodeId": "57b15c51450be454180fa460"
    },
    {
      "type": "node",
      "action": "discovered|updated",
    }
  ]
}
```

Web Hook APIs

Create a new hook

```
POST /api/2.0/hooks
{
  "url": "http://www.abc.com/def"
}
```

Delete an existing hook

```
DELETE /api/2.0/hooks/:id
```

Get a list of hooks

```
GET /api/2.0/hooks
```

Get details of a single hook

```
GET /api/2.0/hooks/:id
```

Update an existing hook

```
PATCH /api/2.0/hooks/:id
{
  "name": "New Hook"
}
```

Redfish Alert Notification

Description

RackHD is enabled to receive redfish based notifications. It is possible to configure a redfish endpoint to send alerts to RackHD. When RackHD receives an alert, it determines which node issued the alert and then it adds some additional context such as nodeId, service tag, etc. Lastly, RackHD publishes the alert to AMQP and Web Hook.

Configuring the Redfish endpoint

If the endpoint is redfish enabled and supports the Redfish EventService, it is possible to configure the endpoint to send the alerts to RackHD. Please note that the “Destination” property in the example below should be a reference to RackHD.

```
POST /redfish/v1/EventService/Subscriptions
{
  "Context": "context string",
  "Description": "Event Subscription Details",
  "Destination": "https://10.240.19.226:8443/api/2.0/notification/alerts",
  "EventTypes": [
    "ResourceAdded",
    "StatusChange",
    "Alert"
  ],
  "Id": "id",
  "Name": "name",
  "Protocol": "Redfish"
}
```

If the node is a Dell node, it is possible to post the Graph.Dell.Configure.Redfish.Alerting workflow. The workflow will:

- 1- Enable Alerts for the Dell node. Equivalent to running “set iDRAC.IPMILan.AlertEnable 1” racadam command.

2- Enable redfish alerts. Equivalent to running “eventfilters set -c idrac.alert.all -a none -n redfish-events” racadam command.

3- Disable the “Audit” info alerts. Equivalent to running “eventfilters set -c idrac.alert.audit.info -a none -n none” racadam command.

The workflow will run the default values if the node’s obm is set and the “rackhdPublicIp” property is set in the rackHD config.json file. Below is an example the default settings:

```
{
  "@odata.context": "/redfish/v1/$metadata#EventDestination.EventDestination",
  "@odata.id": "/redfish/v1/EventService/Subscriptions/b50106d4-32c6-11e7-8b05-64006ac35232",
  "@odata.type": "#EventDestination.v1_0_2.EventDestination",
  "Context": "RackHD Subscription",
  "Description": "Event Subscription Details",
  "Destination": "https://10.1.1.1:8443/api/2.0/notification/alerts",
  "EventTypes": [
    "ResourceAdded",
    "StatusChange",
    "Alert"
  ],
  "EventTypes@odata.count": 3,
  "Id": "b50106d4-32c6-11e7-8b05-64006ac35232",
  "Name": "EventSubscription b50106d4-32c6-11e7-8b05-64006ac35232",
  "Protocol": "Redfish"
}
```

It is possible to overwrite any of the values by adding it to payload when posting the Graph.Configure.Redfish.Alerting workflow. Here is an instance of the payload:

```
{
  "options": {
    "redfish-subscription": {
      "url": "https://10.240.19.130/redfish/v1/EventService/Subscriptions",
      "credential": {
        "username": "root",
        "password": "1234567"
      },
      "data": {
        "Context": "context string",
        "Description": "Event Subscription Details",
        "Destination": "https://1.1.1.1:8443/api/2.0/notification/alerts",
        "EventTypes": [
          "StatusChange",
          "Alert"
        ],
        "Id": "id",
        "Name": "name",
        "Protocol": "Redfish"
      }
    }
  }
}
```

Alert message

In addition to the redfish alert message, RackHD adds the following properties: “sourceIpAddress” (of the BMC), “nodeId”, “macAddress” (of the BMC), “ChassisName”, “ServiceTag”, “SN”.

```
{
  "type": "node",
  "action": "alerts",
  "data": {
    "Context": "context string",
    "EventId": "8689",
    "EventTimestamp": "2017-04-03T10:07:32-0500",
    "EventType": "Alert",
    "MemberId": "7e675c8e-127a-11e7-9fc8-64006ac35232",
    "Message": "The coin cell battery in CMC 1 is not working.",
    "MessageArgs": ["1"],
    "MessageArgs@odata.count": 1,
    "MessageId": "CMC8572",
    "Severity": "Critical",
    "sourceIpAddress": "10.240.19.130",
    "nodeId": "58d94cec316779d4126be134",
    "sourceMacAddress": "64:00:6a:c3:52:32",
    "ChassisName": "PowerEdge R630",
    "ServiceTag": "4666482",
    "SN": "CN747515A80855"
  },
  "severity": "critical",
  "typeId": "58d94cec316779d4126be134",
  "version": "1.0",
  "createdAt": "2017-04-03T14:11:46.245Z"
}
```

AMQP

The messages are pulished to:

- Exchange: **on.events**
- Routing Key: **node.alerts.<severity>.<typeId>.<nodeId>**

Nodes and Catalogs

Nodes are the elements that RackHD manages - compute servers, switches, etc. Nodes typically have at least one catalog, and can have [Pollers](#) and [Workflow Graphs](#) assigned to or working against that node.

Catalogs are free form data structures with information about the nodes. Catalogs are created during ‘discovery’ workflows, and present information that can be requested via API and is available to workflows to operate against.

Defining Nodes

Nodes are defined via a JSON definition that conform to this schema:

- id (string): unique identifier for the node
- type (string): a human readable name for the graph

- name (string): a unique name used by the system and the API to refer to the graph
- autodiscover (boolean):
- sku (string): the SKU 'id' that has been matched from the SKU workflow task
- createdAt (string): ISO8601 date string of time resource was created
- updatedAt (string): ISO8601 date string of time resource was last updated
- identifiers (array of strings): a list of strings that make up alternative identifiers for the node
- obms (array of objects): a list of objects that define out-of-band management access mechanisms
- relations (array of objects): a list of relationship objects

Defining Catalogs

- id (string): unique identifier for the node
- createdAt (string): ISO8601 date string of time resource was created
- updatedAt (string): ISO8601 date string of time resource was last updated
- data (json): A JSON data structure specific to the catalog tool
- node (string): the node to which this catalog is associated
- source (string): type of the data

API Commands for Nodes

The following are common API commands that can be used when running the *on-http* process.

Get Nodes

```
GET /api/current/nodes
```

```
curl <server>/api/current/nodes
```

Get Specific Node

```
GET /api/current/nodes/<id>
```

```
curl <server>/api/current/nodes/<id>
```

Sample switch node after Discovery

```
{
  "type": "switch",
  "name": "nodeName",
  "autoDiscover": true,
  "service": "snmp-ibm-service",
  "config": {
    "host": "10.1.1.3"
  },
  "createdAt": "2015-07-27T22:03:45.353Z",
  "updatedAt": "2015-07-27T22:03:45.353Z",
  "id": "55b6aac1024fd1b349afc145"
}
```

Sample compute node after Discovery

```
{
  "autoDiscover": false,
  "catalogs": [],
  "createdAt": "2015-11-30T21:37:18.441Z",
  "id": "565cc18ec3f522fe51620fa2",
  "identifiers": [
    "08:00:27:27:eb:12"
  ],
  "name": "08:00:27:27:eb:12",
  "obms": [
    {
      "ref": "/api/2.0/obms/58806bb776fab9d82b831e52",
      "service": "noop-obm-service"
    }
  ],
  "relations": [
    {
      "relationType": "enclosedBy",
      "targets": [
        "565cc1d2807f92fc51a7c9c5"
      ]
    }
  ],
  "sku": "565cb91669aa70ab450da9dd",
  "type": "compute",
  "updatedAt": "2015-11-30T21:38:26.755Z",
  "workflows": []
}
```

List all the (latest) catalog data associated with a node

```
GET /api/current/nodes/<id>/catalogs
```

```
curl <server>/api/current/nodes<id>/catalogs
```

To retrieve a specific catalog source for a node

```
GET /api/current/nodes/<id>/catalogs/<source>
```

```
curl <server>/api/current/nodes<id>/catalogs/<source>
```

Sample Output:

```
{
  "createdAt": "2015-11-30T21:37:49.696Z",
  "data": {
    "BIOS Information": {
      "Address": "0xE0000",
      "Characteristics": [
        "ISA is supported",
        "PCI is supported",
        "Boot from CD is supported",
        "Selectable boot is supported",
        "8042 keyboard services are supported (int 9h)",
        "CGA/mono video services are supported (int 10h)",
        "ACPI is supported"
      ],
      "ROM Size": "128 kB",
      "Release Date": "12/01/2006",
    }
  }
}
```

```

    "Runtime Size": "128 kB",
    "Vendor": "innotek GmbH",
    "Version": "VirtualBox"
  },
  "Base Board Information": {
    "Asset Tag": "Not Specified",
    "Chassis Handle": "0x0003",
    "Contained Object Handles": "0",
    "Features": [
      "Board is a hosting board"
    ],
    "Location In Chassis": "Not Specified",
    "Manufacturer": "Oracle Corporation",
    "Product Name": "VirtualBox",
    "Serial Number": "0",
    "Type": "Motherboard",
    "Version": "1.2"
  },
  "Chassis Information": {
    "Asset Tag": "Not Specified",
    "Boot-up State": "Safe",
    "Lock": "Not Present",
    "Manufacturer": "Oracle Corporation",
    "Power Supply State": "Safe",
    "Security Status": "None",
    "Serial Number": "Not Specified",
    "Thermal State": "Safe",
    "Type": "Other",
    "Version": "Not Specified"
  },
  "Inactive": [
    {},
    {},
    {}
  ],
  "OEM Strings": {
    "String 1": "vboxVer_5.0.10",
    "String 2": "vboxRev_104061"
  },
  "OEM-specific Type": {
    "Header and Data": [
      "80 08 08 00 E7 7D 21 00"
    ]
  },
  "System Information": {
    "Family": "Virtual Machine",
    "Manufacturer": "innotek GmbH",
    "Product Name": "VirtualBox",
    "SKU Number": "Not Specified",
    "Serial Number": "0",
    "UUID": "992DA874-C028-4CDD-BB06-C86D525A7056",
    "Version": "1.2",
    "Wake-up Type": "Power Switch"
  }
},
"id": "565cc1ad807f92fc51a7c9bfb",
"node": "565cc18ec3f522fe51620fa2",
"source": "dmi",

```

```
"updatedAt": "2015-11-30T21:37:49.696Z"
}
```

Out of Band Management Settings

Get list of Out of Band Management settings that have been associated with nodes.

Get list of OBM's settings

```
GET /api/current/obms
```

```
curl <server>/api/current/obms
```

Get list of OBM's schemas showing required properties to create an OBM

```
GET /api/current/obms/definitions
```

```
curl <server>/api/current/obms/definitions
```

Create or update a single OBM service and associate it with a node

```
PUT /api/current/obms
```

```
curl -X PUT -H "Content-Type: application/json" -d '{ "nodeId": <node id>, "service": "ipmi-obm-service" }
```

Example output of PUT

```
{
  "id": "5911fa6447f8b7b207f9a485",
  "node": "/api/2.0/nodes/590cbcbf29ba9e40471c9f3c",
  "service": "ipmi-obm-service",
  "config": {
    "user": "admin",
    "host": "172.31.128.2"
  }
}
```

Get a specific OBM setting

```
GET /api/current/obms/<id>
```

```
curl <server>/api/current/obms/<id>
```

PATCH an OBM setting

```
PATCH /api/current/obms/<id>
```

```
curl -X PUT -H "Content-Type: application/json" -d '{ "nodeId": <node id>, "service": "ipmi-obm-service" }
```

Delete an OBM setting

```
DELETE /api/current/obms/<id>
```

```
curl -X DELETE <server>/api/current/obms/<id>
```

To set a no-op OBM setting on a node

```
curl -X PUT -H "Content-Type: application/json" localhost/api/current/nodes/5542b78c130198aa216da3ac -
```

To set a IPMI OBM setting on a node

```
curl -X PUT -H 'Content-Type: application/json' -d '{ "service": "ipmi-obm-service", "config": { "h
```

How to use obms when more than one obm are present on a node

Example: when update firmware workflow is called on a node that has multiple obms (ipmi-obm-service, redfish-obm-service), the payload needs to call out what obm service to use for certain tasks within the workflow that use the obm service..

```
POST /api/current/nodes/<id>/nodes/workflows?name=Graph.Dell.Racadm.Update.Firmware
```

```
{
  "options": {
    "defaults": {
      "filePath": "xyz",
      "serverUsername": "abc",
      "serverPassword": "123",
      "serverFilePath": "def"
    },
    "set-boot-pxe": {
      "obmServiceName": "ipmi-obm-service"
    },
    "reboot": {
      "obmServiceName": "ipmi-obm-service"
    }
  }
}
```

In Band Management Settings

Get list of In Band Management settings that have been associated with nodes.

Get list of IBMs settings

```
GET /api/current/ibms
```

```
curl <server>/api/current/ibms
```

Get list of IBMs schemas showing required properties to create an IBM

```
GET /api/current/ibms/definitions
```

```
curl <server>/api/current/ibms/definitions
```

Create or update a single IBM service and associate it with a node

```
PUT /api/current/ibms
```

```
curl -X PUT -H "Content-Type: application/json" -d '{ "nodeId": <node id>, "service": "snmp-ibm-serv
```

Example output of PUT

```
{
  "id": "591c569c087752c67428e4b3",
  "node": "/api/2.0/nodes/590cbcbf29ba9e40471c9f3c",
  "service": "snmp-ibm-service",
  "config": {
    "host": "172.31.128.2"
  }
}
```

Get a specific IBM setting

```
GET /api/current/ibms/<id>
```

```
curl <server>/api/current/ibms/<id>
```

PATCH an IBM setting

```
PATCH /api/current/ibms/<id>
```

```
curl -X PUT -H "Content-Type: application/json" -d '{ "nodeId": <node id>, "service": "snmp-ibm-serv
```

Delete an IBM setting

```
DELETE /api/current/ibms/<id>
```

```
curl -X DELETE <server>/api/current/ibms/<id>
```

Node Tags

Add a tag to a node

```
PATCH /api/current/nodes/<id>/tags
```

```
curl -H "Content-Type: application/json" -X PATCH -d '{ "tags": [<list of tags>]}' <server>/api/curre
```

List tags for a node

```
GET /api/current/nodes/<id>/tags
```

```
curl <server>/api/current/nodes/<id>/tags
```

Delete a tag from a node

```
DELETE /api/current/nodes/<id>/tags/<tagname>
```

```
curl -X DELETE <server>/api/current/nodes/<id>/tags/<tagname>
```

Node Relations

List relations for a node

```
GET <server>/api/current/nodes/<id>/relations
```

```
curl <server>/api/current/nodes/<id>/relations
```

Sample response:

```
[
  {
    "relationType": "contains",
    "targets": [
      "57c0d980851053795fdc7bcf",
      "57c0d6bd851053795fdc7bc4"
    ]
  }
]
```


Add relations to a node

```
PUT <server>/api/current/nodes/<id>/relations
```

```
curl -H "Content-Type: application/json" -X PUT -d '{ <relationType>: [<list of targets>]}' <server>
```

Sample request body:

```
{
  "contains": [ "57c0d980851053795fdc7bcf", "57c0d6bd851053795fdc7bc4" ]
}
```

Sample response body:

```
[
  {
    "autoDiscover": false,
    "createdAt": "2016-08-30T18:39:57.819Z",
    "name": "demoRack",
    "relations": [
      {
        "relationType": "contains",
        "targets": [
          "57c0d980851053795fdc7bcf",
          "57c0d6bd851053795fdc7bc4"
        ]
      }
    ],
    "tags": [],
    "type": "rack",
    "updatedAt": "2016-08-30T21:07:11.717Z",
    "id": "57c5d2fd64bda4e679146530"
  },
  {
    "autoDiscover": false,
    "createdAt": "2016-08-27T00:06:24.784Z",
    "identifiers": [
      "08:00:27:10:1f:25"
    ],
    "name": "08:00:27:10:1f:25",
    "relations": [
      {
        "relationType": "containedBy",
        "targets": [
          "57c5d2fd64bda4e679146530"
        ]
      }
    ],
    "sku": null,
    "tags": [],
    "type": "compute",
    "updatedAt": "2016-08-30T21:07:11.729Z",
    "id": "57c0d980851053795fdc7bcf"
  },
  {
    "autoDiscover": false,
    "createdAt": "2016-08-26T23:54:37.249Z",
    "identifiers": [
      "08:00:27:44:97:79"
    ],
    "name": "08:00:27:44:97:79",
    "relations": [
      {
        "relationType": "containedBy",
        "targets": [
          "57c5d2fd64bda4e679146530"
        ]
      }
    ],
    "sku": null,
    "tags": [],
    "type": "compute",
    "updatedAt": "2016-08-30T21:07:11.729Z",
    "id": "57c0d980851053795fdc7bcf"
  }
]
```

```
    "name": "08:00:27:44:97:79",
    "relations": [
      {
        "relationType": "containedBy",
        "targets": [
          "57c5d2fd64bda4e679146530"
        ]
      }
    ],
    "sku": null,
    "tags": [],
    "type": "compute",
    "updatedAt": "2016-08-30T21:07:11.724Z",
    "id": "57c0d6bd851053795fdc7bc4"
  }
]
```

Remove Relations from a node

```
DELETE <server>/api/current/nodes/<id>/relations
```

```
curl -H "Content-Type: application/json" -X DELETE -d '{ <relationType>: [<list of targets>]}' <server>
```

Sample request body:

```
{
  "contains": [ "57c0d980851053795fdc7bcf", "57c0d6bd851053795fdc7bc4" ]
}
```

Sample response body:

```
[
  {
    "autoDiscover": false,
    "createdAt": "2016-08-30T18:39:57.819Z",
    "name": "demoRack",
    "relations": [],
    "tags": [],
    "type": "rack",
    "updatedAt": "2016-08-30T21:14:11.553Z",
    "id": "57c5d2fd64bda4e679146530"
  },
  {
    "autoDiscover": false,
    "createdAt": "2016-08-27T00:06:24.784Z",
    "identifiers": [
      "08:00:27:10:1f:25"
    ],
    "name": "08:00:27:10:1f:25",
    "relations": [],
    "sku": null,
    "tags": [],
    "type": "compute",
    "updatedAt": "2016-08-30T21:14:11.566Z",
    "id": "57c0d980851053795fdc7bcf"
  },
  {
    "autoDiscover": false,
    "createdAt": "2016-08-26T23:54:37.249Z",
```

```

    "identifiers": [
        "08:00:27:44:97:79"
    ],
    "name": "08:00:27:44:97:79",
    "relations": [],
    "sku": null,
    "tags": [],
    "type": "compute",
    "updatedAt": "2016-08-30T21:14:11.559Z",
    "id": "57c0d6bd851053795fdc7bc4"
  }
]

```

Workflows

Workflow Graphs

The workflow graph definition specifies the order in which tasks should run and provides any context and/or option values to pass to these functions.

Complex graphs may define event-based tasks or specify data/event channels that should exist between concurrently-run tasks.

Defining Graphs

Graphs are defined via a JSON definition that conform to this schema:

- friendlyName (string): a human readable name for the graph
- injectableName (string): a unique name used by the system and the API to refer to the graph
- tasks (array of objects): a list of task definitions or references to task definitions.
 - tasks.label (string): a unique string to be used as a reference within the graph definition
 - tasks.[taskName] (string): the injectableName of a task in the database to run. This or taskDefinition is required.
 - tasks.[taskDefinition] (object): an inline definition of a task, instead of one in the database. This or taskName is required.
 - tasks.[ignoreFailure] (boolean): ignoreFailure: true will prevent the graph from failing on task failure
 - tasks.[waitOn] (object): key/value pairs referencing other task labels to desired states of those tasks to trigger running on. Available states are *succeeded*, *failed* and *finished* (run on succeeded or failed). If waitOn is not specified, the task will run on graph start.
- [options]
 - options.[defaults] (object): key, value pairs that will be handed to any tasks that have matching option keys
 - options.<label> (object): key, value pairs that should all be handed to a specific task

Graph definition attributes

Graph Tasks

The *tasks* field in a graph definition represents the collection of tasks that make up the runtime behavior of the graph. The task definition is referenced by the `taskName` field (which maps to the `injectableName` field in the task definition). The `label` field is used as a reference when specifying dependencies for other tasks in the graph definition. For example, this graph will run three tasks one after the other:

```
{
  "injectableName": "Graph.Example.Linear",
  "friendlyName": "Linear ordered tasks",
  "tasks": [
    {
      "label": "task-1",
      "taskName": "Task.example"
    },
    {
      "label": "task-2",
      "taskName": "Task.example",
      "waitOn": {
        "task-1": "succeeded"
      }
    },
    {
      "label": "task-3",
      "taskName": "Task.example",
      "waitOn": {
        "task-2": "succeeded"
      }
    }
  ]
}
```

The ordering is specified by the `waitOn` key in each task object, which specifies conditions that must be met before each task can be run. In the above graph definition, `task-1` has no dependencies, so it will be run immediately, `task-2` has a dependency on `task-1` succeeding, and `task-3` has a dependency on `task-2` succeeding.

Here is an example of a graph that will run tasks in parallel:

```
{
  "injectableName": "Graph.Example.Parallel",
  "friendlyName": "Parallel ordered tasks",
  "tasks": [
    {
      "label": "task-1",
      "taskName": "Task.example"
    },
    {
      "label": "task-2",
      "taskName": "Task.example",
      "waitOn": {
        "task-1": "succeeded"
      }
    },
    {
      "label": "task-3",
      "taskName": "Task.example",
      "waitOn": {
        "task-1": "succeeded"
      }
    }
  ]
}
```

```
}

```

This graph is almost the same as the “Linear ordered tasks” example, except that `task-2` and `task-3` *both* have a dependency on `task-1`. When `task-1` succeeds, `task-2` and `task-3` will be started in parallel.

Tasks can also be ordered based on multiple dependencies:

```
{
  "injectableName": "Graph.Example.MultipleDependencies",
  "friendlyName": "Tasks with multiple dependencies",
  "tasks": [
    {
      "label": "task-1",
      "taskName": "Task.example"
    },
    {
      "label": "task-2",
      "taskName": "Task.example"
    },
    {
      "label": "task-3",
      "taskName": "Task.example",
      "waitOn": {
        "task-1": "succeeded",
        "task-2": "succeeded"
      }
    }
  ]
}
```

In the above example, `task-1` and `task-2` will be started in parallel, and `task-3` will only be started once `task-1` and `task-2` have both succeeded. **Graph Options**

As detailed in the [Task Definitions](#) section, each task definition has an options object that can be used to customize the task. All values set in the options objects are considered defaults, and can be overridden within the Graph definition. Additionally, the options values can be overridden again by the data in the API request made to run the graph.

For example, a simple task definition with options looks like this:

```
{
  "injectableName": "Task.Example.Options",
  "friendlyName": "Task with basic options",
  "implementsTask": "Task.Base.Example",
  "options": {
    "option1": "value 1",
    "option2": "value 2"
  },
  "properties": {}
}
```

As is, this task definition specifies default values of “value 1” and “value 2” for its respective options. In the graph definition, these values can be changed to have new defaults by adding a key to the `Graph.options` object that matches the label string given to the task object (“example-options-task” in this case):

```
{
  "injectableName": "Graph.Example.Options",
  "friendlyName": "Override options for a task",
  "options": {
    "example-options-task": {
      "option1": "overridden value 1",

```

```
        "option2": "overridden value 2"
      }
    },
    "tasks": [
      {
        "label": "example-options-task",
        "taskName": "Task.Example.Options"
      }
    ]
  }

// Task.Example.Options will be run as this
{
  "injectableName": "Task.Example.Options",
  "friendlyName": "Task with basic options",
  "implementsTask": "Task.Base.Example",
  "options": {
    "option1": "overridden value 1",
    "option2": "overridden value 2"
  },
  "properties": {}
}
```

Sometimes, it is necessary to be able to propagate the same values to multiple tasks, but it can be a chore to make a separate options object for each task label. In this case, there is a special field used in the `Graph.options` object called `defaults`. When `defaults` is set, the graph will iterate through each key in the object and override that value for every task definition that also has that key in its respective options object. In the above example, the `Task.Example.Options` definition will be changed with new values for `option1` and `option2`, but not for `option3`, since `option3` does not exist in the options object for that task definition:

```
{
  "injectableName": "Graph.Example.Defaults",
  "friendlyName": "Override options with defaults",
  "options": {
    "defaults": {
      "option1": "overridden value 1",
      "option2": "overridden value 2",
      "option3": "this will not get set"
    }
  },
  "tasks": [
    {
      "label": "example-options-task",
      "taskName": "Task.Example.Options"
    }
  ]
}

// Task.Example.Options will be run as this
{
  "injectableName": "Task.Example.Options",
  "friendlyName": "Task with basic options",
  "implementsTask": "Task.Base.Example",
  "options": {
    "option1": "overridden value 1",
    "option2": "overridden value 2"
  },
  "properties": {}
}
```

```
}
```

The `defaults` object can be used to share values across every task definition that includes them, as in this example workflow that validates and sets a username.

```
{
  "injectableName": "Graph.Example.SetUsername",
  "friendlyName": "Set a username",
  "options": {
    "defaults": {
      "username": "TESTUSER",
      "group": "admin"
    }
  },
  "tasks": [
    {
      "label": "validate-username",
      "taskName": "Task.Example.ValidateUsername"
    },
    {
      "label": "set-username",
      "taskName": "Task.Example.SetUsername",
      "waitOn": {
        "validate-username": "succeeded"
      }
    }
  ]
}

// Task.Example.ValidateUsername definition
{
  "injectableName": "Task.Example.Validateusername",
  "friendlyName": "Validate a username",
  "implementsTask": "Task.Base.ValidateUsername",
  "options": {
    "username": null,
  },
  "properties": {}
}

// Task.Example.SetUsername definition
{
  "injectableName": "Task.Example.Setusername",
  "friendlyName": "Set a username",
  "implementsTask": "Task.Base.SetUsername",
  "options": {
    "username": null,
    "group": null
  },
  "properties": {}
}
```

Both tasks will share the “TESTUSER” value for the `username` option, but only the `Task.Example.SetUsername` task will use the value for `group`, since it is the only task definition in this graph with that key in its options object. After processing the graph definition and the default options, the task definitions will be run as:

```
// Task.Example.ValidateUsername definition after Graph defaults applied
{
  "injectableName": "Task.Example.Validateusername",
  "friendlyName": "Validate a username",
  "implementsTask": "Task.Base.ValidateUsername",
  "options": {
    "username": "TESTUSER"
  },
  "properties": {}
}

// Task.Example.SetUsername definition after Graph defaults applied
{
  "injectableName": "Task.Example.Setusername",
  "friendlyName": "Set a username",
  "implementsTask": "Task.Base.SetUsername",
  "options": {
    "username": "TESTUSER",
    "group": "admin"
  },
  "properties": {}
}
```

API Commands for Graphs

The following are API commands that can be used when running the *on-http* process.

Get Available Graphs in the Library

```
GET /api/current/workflows/graphs
```

```
curl <server>/api/current/workflows/graphs
```

Deprecated 1.1 API - Get Available Graphs in the Library

```
GET /api/1.1/workflows/library/*
```

```
curl <server>/api/1.1/workflows/library/*
```

Query the State of an Active Graph

```
GET /api/current/nodes/<id>/workflows?active=true
```

```
curl <server>/api/current/workflows?active=true
```

Deprecated 1.1 API - Query State of an Active Graph

```
GET /api/1.1/nodes/<id>/workflows/active
```

```
curl <server>/api/1.1/nodes/<id>/workflows/active
```

Cancel or Kill an Active Graph running against a Node

```
PUT /api/current/nodes/<id>/workflows/action
{
  "command": "cancel"
}
```



```
curl -X PUT \
  -H 'Content-Type: application/json' \
  -d '{"command": "cancel"}' \
  <server>/api/current/nodes/<id>/workflows/action
```

Deprecated 1.1 API - Cancel or Kill an Active Graph running against a Node

```
DELETE /api/1.1/nodes/<id>/workflows/active
```

```
curl -X DELETE <server>/api/1.1/nodes/<id>/workflows/active
```

List all Graphs that have or are running against a Node

```
GET /api/current/nodes/<id>/workflows
```

```
curl <server>/api/current/nodes/<id>/workflows
```

Create a Graph Definition

```
PUT /api/current/workflows/graphs
{
  <json definition of graph>
}
```

Deprecated 1.1 API - Create a Graph Definition

```
PUT /api/1.1/workflows
{
  <json definition of graph>
}
```

Run a New Graph Against a Node

Find the graph definition you would like to use and copy the top-level *injectableName* attribute.

```
POST /api/current/nodes/<id>/workflows
{
  "name": <graph name>
}
```

```
curl -X POST -H 'Content-Type: application/json' <server>/api/current/nodes/<id>/workflows?name=<graph name>
OR
curl -X POST \
  -H 'Content-Type: application/json' \
  -d '{"name": "<graphname>"}' \
  <server>/api/current/nodes/<id>/workflows
```

To override option values, add an options object to the POST data as detailed in the [Graph Options](#) section.

```
POST /api/current/nodes/<id>/workflows
{
  "name": <graph name>
  "options": { <graph options here> }
}
```

For example, to override an option “username” for all tasks in a graph that utilize that option (see [Graph Username Example](#), send the following request:

```
POST /api/current/nodes/<id>/workflows
{
```

```
"name": <graph name>
"options": {
  "defaults": {
    "username": "customusername"
  }
}
}
```

Sample Output:

```
{
  "_events": {},
  "_status": "valid",
  "cancelled": false,
  "completeEventString": "complete",
  "context": {
    "b9b29b18-309f-439d-8de7-a1042c400d9a": {
      "cancelled": false,
      "local": {
        "stats": {}
      },
      "parent": {}
    },
    "graphId": "c2d48e40-7beb-4d64-9d59-a475c6732780",
    "target": "54daab331ee7cb79d888cba5"
  },
  "createdAt": "2015-02-11T18:35:25.277Z",
  "definition": {
    "friendlyName": "Zerotouch vEOS Graph",
    "injectableName": "Graph.Arista.Zerotouch.vEOS",
    "options": {},
    "tasks": [
      {
        "label": "zerotouch-veos",
        "taskDefinition": {
          "friendlyName": "Arista Zerotouch vEOS",
          "implementsTask": "Task.Base.Arista.Zerotouch",
          "injectableName": "Task.Inline.Arista.Zerotouch.vEOS",
          "options": {
            "bootConfig": "arista-boot-config",
            "bootfile": "zerotouch-vEOS.swi",
            "eosImage": "zerotouch-vEOS.swi",
            "hostname": "MonorailVEOS",
            "profile": "zerotouch-configure.zt",
            "startupConfig": "arista-startup-config"
          },
          "properties": {
            "os": {
              "switch": {
                "type": "eos",
                "virtual": true
              }
            }
          }
        }
      }
    ]
  },
  "failedStates": [
```

```

    "failed",
    "timeout",
    "cancelled"
  ],
  "finishedStates": [
    "failed",
    "succeeded",
    "timeout",
    "cancelled"
  ],
  "finishedTasks": [],
  "id": "54dba0edc44e16c9164110a3",
  "injectableName": "Graph.Arista.Zerotouch.vEOS",
  "instanceId": "c2d48e40-7beb-4d64-9d59-a475c6732780",
  "name": "Zerotouch vEOS Graph",
  "pendingTasks": [
    {
      "cancelled": false,
      "context": {
        "cancelled": false,
        "local": {
          "stats": {}
        },
        "parent": {}
      },
      "definition": {
        "friendlyName": "Arista Zerotouch vEOS",
        "implementsTask": "Task.Base.Arista.Zerotouch",
        "injectableName": "Task.Inline.Arista.Zerotouch.vEOS",
        "options": {
          "bootConfig": "arista-boot-config",
          "bootfile": "zerotouch-vEOS.swi",
          "eosImage": "zerotouch-vEOS.swi",
          "hostname": "MonorailVEOS",
          "profile": "zerotouch-configure.zt",
          "startupConfig": "arista-startup-config"
        },
        "properties": {
          "os": {
            "switch": {
              "type": "eos",
              "virtual": true
            }
          }
        },
        "runJob": "Job.Arista.Zerotouch"
      },
      "dependents": [],
      "failedStates": [
        "failed",
        "timeout",
        "cancelled"
      ],
      "friendlyName": "Arista Zerotouch vEOS",
      "ignoreFailure": false,
      "instanceId": "b9b29b18-309f-439d-8de7-a1042c400d9a",
      "name": "Task.Inline.Arista.Zerotouch.vEOS",
      "options": {

```

```
    "bootConfig": "arista-boot-config",
    "bootfile": "zerotouch-vEOS.swi",
    "eosImage": "zerotouch-vEOS.swi",
    "hostname": "MonorailVEOS",
    "profile": "zerotouch-configure.zt",
    "startupConfig": "arista-startup-config"
  },
  "parentContext": {
    "b9b29b18-309f-439d-8de7-a1042c400d9a": {
      "cancelled": false,
      "local": {
        "stats": {}
      },
      "parent": {}
    },
    "graphId": "c2d48e40-7beb-4d64-9d59-a475c6732780",
    "target": "54daab331ee7cb79d888cba5"
  },
  "properties": {
    "os": {
      "switch": {
        "type": "eos",
        "virtual": true
      }
    }
  },
  "retriesAllowed": 5,
  "retriesAttempted": 0,
  "state": "pending",
  "stats": {
    "completed": null,
    "created": "2015-02-11T18:35:25.269Z",
    "started": null
  },
  "successStates": [
    "succeeded"
  ],
  "tags": [],
  "waitingOn": []
},
"ready": [],
"serviceGraph": null,
"tasks": {
  "b9b29b18-309f-439d-8de7-a1042c400d9a": {
    "cancelled": false,
    "context": {
      "cancelled": false,
      "local": {
        "stats": {}
      },
      "parent": {}
    },
    "definition": {
      "friendlyName": "Arista Zerotouch vEOS",
      "implementsTask": "Task.Base.Arista.Zerotouch",
      "injectableName": "Task.Inline.Arista.Zerotouch.vEOS",
      "options": {
```

```

        "bootConfig": "arista-boot-config",
        "bootfile": "zerotouch-vEOS.swi",
        "eosImage": "zerotouch-vEOS.swi",
        "hostname": "MonorailVEOS",
        "profile": "zerotouch-configure.zt",
        "startupConfig": "arista-startup-config"
    },
    "properties": {
        "os": {
            "switch": {
                "type": "eos",
                "virtual": true
            }
        }
    },
    "runJob": "Job.Arista.Zerotouch"
},
"dependents": [],
"failedStates": [
    "failed",
    "timeout",
    "cancelled"
],
"friendlyName": "Arista Zerotouch vEOS",
"ignoreFailure": false,
"instanceId": "b9b29b18-309f-439d-8de7-a1042c400d9a",
"name": "Task.Inline.Arista.Zerotouch.vEOS",
"options": {
    "bootConfig": "arista-boot-config",
    "bootfile": "zerotouch-vEOS.swi",
    "eosImage": "zerotouch-vEOS.swi",
    "hostname": "MonorailVEOS",
    "profile": "zerotouch-configure.zt",
    "startupConfig": "arista-startup-config"
},
"parentContext": {
    "b9b29b18-309f-439d-8de7-a1042c400d9a": {
        "cancelled": false,
        "local": {
            "stats": {}
        },
        "parent": {}
    },
    "graphId": "c2d48e40-7beb-4d64-9d59-a475c6732780",
    "target": "54daab331ee7cb79d888cba5"
},
"properties": {
    "os": {
        "switch": {
            "type": "eos",
            "virtual": true
        }
    }
},
"retriesAllowed": 5,
"retriesAttempted": 0,
"state": "pending",
"stats": {

```

```
        "completed": null,
        "created": "2015-02-11T18:35:25.269Z",
        "started": null
      },
      "successStates": [
        "succeeded"
      ],
      "tags": [],
      "waitingOn": []
    }
  },
  "updatedAt": "2015-02-11T18:35:25.277Z"
}
```

Workflow Examples

Creating a Custom Zerotouch Graph for Arista

This section provides instructions for creating a custom zerotouch graph for Arista machines, including defining a custom EOS image, custom startup-config, and custom zerotouch script.

Below is an example zerotouch graph for booting a vEOS (virtual arista) machine. It uses an inline task definition (as opposed to creating a new task definition as a separate step):

```
{
  friendlyName: 'Zerotouch vEOS Graph',
  injectableName: 'Graph.Arista.Zerotouch.vEOS',
  tasks: [
    {
      label: 'zerotouch-veos',
      taskDefinition: {
        friendlyName: 'Arista Zerotouch vEOS',
        injectableName: 'Task.Inline.Arista.Zerotouch.vEOS',
        implementsTask: 'Task.Base.Arista.Zerotouch',
        options: {
          profile: 'zerotouch-configure.zt',
          bootConfig: 'arista-boot-config',
          startupConfig: 'arista-startup-config',
          eosImage: 'common/zerotouch-vEOS.swi',
          bootfile: 'zerotouch-vEOS.swi',
          hostname: 'MonorailVEOS'
        },
        properties: {
          os: {
            switch: {
              type: 'vEOS',
              virtual: true
            }
          }
        }
      }
    }
  ]
}
```

To customize this graph, change the following fields:

Field	Description
friendlyName	A unique friendly name for the graph.
injectableName	A unique injectable name for the graph.
task/friendlyName	A unique friendlyName for the task.
task/injectableName	A unique injectableName for the task.
profile	The default profile is sufficient for most cases. See the Zerotouch Profile section for more information.
bootConfig	The default bootConfig is sufficient for most cases. See the Zerotouch Profile section for more information.
startupConfig	Specify the name of the custom startup config. See the Adding Zerotouch Templates section for more information.
eosImage	Specify the name of the EOS image. See the Adding EOS Images section for more information.
bootfile	In most cases, specify the eosImage name.
hostname	An value rendered into the default arista-startup-config template. Depending on the template, this may be optional.
properties	A object containing any tags/metadata that you wish to add.

Adding Zerotouch Templates

Creation

Templates are defined using `ejs` syntax. To define template variables, use this syntax:

```
<%=variableName%>
```

In order to provide a value for this variable when the template is rendered, add the variable name as a key in the options object of the custom zerotouch task definition:

```
taskDefinition: {
  <other values>
  options: {
    hostname: 'CustomHostName'
  }
}
```

The above code renders the following startup config as shown here:

```
Unrendered:
!
hostname <%=hostname%>
!
```

```
Rendered:
!
hostname CustomHostName
!
```

Uploading

To upload a template, use the templates API:

```
PUT /api/current/templates/library/<filename>
Content-Type: text/plain
```

```
curl -X PUT \
  -H 'Content-Type: text/plain' \
  -d "<startup config template>" \
  <server>/api/current/templates/library/<filename>
```

Deprecated 1.1 API - To upload a template, use the templates API:

```
PUT /api/1.1/templates/library/<filename>
Content-Type: application/octet-stream
```

```
curl -X PUT \
  -H 'Content-Type: application/octet-stream' \
  -d "<startup config template>" \
  <server>/api/1.1/templates/library/<filename>
```

Adding EOS Images

Move any EOS images you would like to use into <on-http directory>/static/http/common/.

In the task options, reference the EOS image name along with the common directory, e.g. eosImage: common/<eosImageName>.

Zerotouch Profile

A zerotouch profile is a script template that is executed by the switch during zerotouch. A basic profile looks like the following:

```
#!/usr/bin/Cli -p2
enable
copy {{ api.templates }}/<%=startupConfig%>?nodeId={{ task.nodeId }} flash:startup-config
copy {{ api.templates }}/<%=bootConfig%>?nodeId={{ task.nodeId }} flash:boot-config
copy http://<%=server%>:<%=port%>/common/<%=eosImage%> flash:
exit
```

Adding `#!/usr/bin/Cli -p2` tells the script to be executed by the Arista's CLI parser. Using `#!/bin/bash` for more control is also an option. If using bash for zerotouch config, any config and imaging files should go into the `/mnt/flash/` directory.

Zerotouch Boot Config

The zerotouch boot config is a very simple config that specifies which EOS image file to boot. This should almost always match the EOS image filename you have provided, e.g.:

```
SWI=flash:/<%=bootfile%>
```

Creating a Linux Commands Graph

Linux Commands Task

The Linux Commands task is a generic task that enables running of any shell commands against a node booted into a microkernel. These commands are specified in JSON objects within the `options.commands` array of the task definition. Optional parameters can be specified to enable cataloging of command output.

A very simple example task definition looks like:

```
{
  "friendlyName" : "Shell commands basic",
  "implementsTask" : "Task.Base.Linux.Commands",
  "injectableName" : "Task.Linux.Commands.BasicExample",
  "options" : {
    "commands" : [
      {
        "command" : "echo testing"
      },
      {

```



```

        "command": "ls"
      }
    ],
    "properties" : { }
  }
}

```

There is an example task included in the monorail system under the name “Task.Linux.Commands” that makes use of all parameters that the task can take:

```

{
  "friendlyName" : "Shell commands",
  "implementsTask" : "Task.Base.Linux.Commands",
  "injectableName" : "Task.Linux.Commands",
  "options" : {
    "commands" : [
      {
        "command" : "sudo ls /var",
        "catalog" : {
          "format" : "raw",
          "source" : "ls var"
        }
      },
      {
        "command" : "sudo lshw -json",
        "catalog" : {
          "format" : "json",
          "source" : "lshw user"
        }
      },
      {
        "command" : "test",
        "acceptedResponseCodes" : [
          1
        ]
      }
    ]
  },
  "properties" : {
    "commands" : {}
  }
}

```

The task above runs three commands and catalogs the output of the first two.

```

sudo ls /var
sudo lshw -json
test

```

Specifying Scripts or Binaries to Download and Run

Some use cases are too complex to be performed by embedding commands in JSON. Using a pre-defined file may be more convenient. You can define a file to download and run by specifying a “downloadUrl” field in addition to the “command” field.

```

"options": {
  "commands" : [
    {
      "command": "bash myscript.sh",

```

```
        "downloadUrl": "{{ api.templates }}/myscript.sh?nodeId={{ task.nodeId }}"
      }
    ]
  }
```

This will cause the command runner script on the node to download the script from the specified route (server:port will be prepended) to the working directory, and execute it according to the specified command (e.g. *bash myscript.sh*). You must specify how to run the script correctly in the command field (e.g. *node myscript.js arg1 arg2, ./myExecutable*).

A note on convention: binary files should be uploaded via the */api/current/files* route, and script templates should be uploaded/downloaded via the */api/current/templates* route.

Defining Script Templates

Scripts can mean simple shell scripts, python scripts, etc.

In many cases, you may need access to variables in the script that can be rendered at runtime. Templates are defined using *ejs* syntax (variables in *<%=variable%>* tags). Variables are rendered based on the option values of task definition, for example, if a task is defined with these options...

```
"options": {
  "foo": "bar",
  "baz": "qux",
  "commands" : [
    {
      "command": "bash myscript.sh",
      "downloadUrl": "{{ api.templates }}/myscript.sh?nodeId={{ task.nodeId }}"
    }
  ]
}
```

...then the following script template...

```
echo <%=foo%>
echo <%=baz%>
```

...is rendered as below when it is run by a node:

```
echo bar
echo qux
```

Predefined template variables

The following variables are predefined and available for use by all templates:

Field	Description
server	This refers to the base IP of the RackHD server
port	This refers to the base port of the RackHD server
ipaddress	This refers to the ipaddress of the requestor
macaddress	This refers to the macaddress, as derived from an IP to MAC lookup, of the requestor
netmask	This refers to the netmask configured for the RackHD DHCP server
gateway	This refers to the gateway configured for the RackHD DHCP server
api	<p>Values used for constructing API requests in a template:</p> <ul style="list-style-type: none"> • server – the base URI for the RackHD http server (e.g. <code>http://<server>:<port></code>) • base – the base http URI for the RackHD api (e.g. <code>http://<server>:<port>/api/current</code>) • templates – the base http URI for the RackHD api files route (e.g. <code>http://<server>:<port>/api/current/templates</code>) • profiles – the base http URI for the RackHD api files route (e.g. <code>http://<server>:<port>/api/current/profiles</code>) • lookups – the base http URI for the RackHD api files route (e.g. <code>http://<server>:<port>/api/current/lookups</code>) • files – the base http URI for the RackHD api files route (e.g. <code>http://<server>:<port>/api/current/files</code>) • nodes – the base http URI for the RackHD api nodes route (e.g. <code>http://<server>:<port>/api/current/nodes</code>)
context	This refers to the shared context object that all tasks in a graph have R/W access to. Templates receive a readonly snapshot of this context when they are rendered.
task	<p>Values used by the currently running task:</p> <ul style="list-style-type: none"> • nodeId – The node identifier that the graph is bound to via the graph context.
sku	This refers to the SKU configuration data fetched from a SKU definition. This field is added automatically if a SKU configuration exists in the the SKU pack, rather than being specified by a user. For more information, please see SKUs
env	This refers to the environment configuration data retrieved from the environment database collection. Similar to sku, this field is added automatically, rather than specified by a user.

Uploading Script Templates

Script templates can be uploaded using the Monorail templates API

```
PUT /api/current/templates/library/<filename>
Content-type: text/plain
---
curl -X PUT -H "Content-Type: text/plain" --data-binary @<script> <server>/api/current/templates/lib
```

Deprecated 1.1 API - Uploading Script Templates

```
PUT /api/1.1/templates/library/<filename>
Content-type: application/octet-stream
---
curl -X PUT -H "Content-Type: application/octet-stream" --data-binary @<script> <server>/api/1.1/temp
```

Uploading Binary Files

Binary executables can be uploaded using the Monorail files API:

```
PUT /api/current/files/<filename>
---
curl -T <binary> <server>/api/current/templates/library/<filename>
```

Available Options for Command JSON Objects

The task definition above makes use of the different options available for parsing and handling of command output. Available options are detailed below:

Name	Type	Required?	Description
command	string	command or script field required	command to run
downloadUrl	string	API route suffix for file download	script/file to download and run
catalog	object	no	an object specifying cataloging parameters if the command output should be cataloged
accept-dResponseCodes	arrayOf-String	no	non-zero exit codes from the command that should not be treated as failures

The catalog object in the above table may look like:

Name	Type	Required?	Description
format	string	yes	The parser to should use for output. Available formats are <i>raw</i> , <i>json</i> , and <i>xml</i> .
source	string	no	What the 'source' key value in the database document should be. Defaults to 'unknown' if not specified.

Creating a Graph with a Custom Shell Commands Task

To use this feature, new workflows and tasks (units of work) must be registered in the system. To create a basic workflow that runs user-specified shell commands with specified images, do the following steps:

1. Define a custom workflow task with the images specified to be used (this is not necessary if you don't need to use a custom image):

```
PUT <server>/api/current/workflows/tasks
Content-Type: application/json
{
  "friendlyName": "Bootstrap Linux Custom",
  "injectableName": "Task.Linux.Bootstrap.Custom",
  "implementsTask": "Task.Base.Linux.Bootstrap",
  "options": {
```

```

        "kernelFile": "vmlinuz-1.2.0-rancher",
        "initrdFile": "initrd-1.2.0-rancher",
        "dockerFile": "discovery.docker.tar.xz",
        "kernelUri": "{{ api.server }}/common/{{ options.kernelFile }}",
        "initrdUri": "{{ api.server }}/common/{{ options.initrdFile }}",
        "dockerUri": "{{ api.server }}/common/{{ options.dockerFile }}",
        "profile": "rancherOS.ipxe",
        "comport": "ttyS0"
    },
    "properties": {}
}

```

2. Define a task that contains the commands to be run, adding or removing command objects below in the `options.commands` array:

```

PUT <server>/api/current/workflows/tasks
Content-Type: application/json
{
    "friendlyName": "Shell commands user",
    "injectableName": "Task.Linux.Commands.User",
    "implementsTask": "Task.Base.Linux.Commands",
    "options": {
        "commands": [    <add command objects here>    ]
    },
    "properties": {"type": "userCreated" }
}

```

The output from the first command (`lshw`) will be parsed as JSON and cataloged in the database under the “`lshw user`” source value. The output from the second command will only be logged, since format and source haven’t been specified. The third command will normally fail, since `test` has an exit code of 1, but in this case we have specified that this is acceptable and not to fail. This feature is useful with certain binaries that have acceptable non-zero exit codes.

Putting it All Together

Now define a custom workflow that combines these tasks and runs them in a sequence. This one is set up to make OBM calls as well.

```

PUT <server>/api/current/workflows/
Content-Type: application/json
{
    "friendlyName": "Shell Commands User",
    "injectableName": "Graph.ShellCommands.User",
    "tasks": [
        {
            "label": "set-boot-pxe",
            "taskName": "Task.Obm.Node.PxeBoot",
            "ignoreFailure": true
        },
        {
            "label": "reboot-start",
            "taskName": "Task.Obm.Node.Reboot",
            "waitOn": {
                "set-boot-pxe": "finished"
            }
        },
        {
            "label": "bootstrap-custom",
            "taskName": "Task.Linux.Bootstrap.Custom",

```

```
        "waitOn": {
            "reboot-start": "succeeded"
        }
    },
    {
        "label": "shell-commands",
        "taskName": "Task.Linux.Commands.User",
        "waitOn": {
            "bootstrap-custom": "succeeded"
        }
    },
    {
        "label": "reboot-end",
        "taskName": "Task.Obm.Node.Reboot",
        "waitOn": {
            "shell-commands": "finished"
        }
    }
]
}
```

With all of these data, the injectableName and friendlyName can be any string value, as long the references to injectableName are consistent across the three JSON documents.

After defining these custom workflows, you can then run one against a node by referencing the injectableName used in the JSON posted to `/api/current/workflows/`:

```
curl -X POST localhost/api/current/nodes/<identifier>/workflows?name=Graph.ShellCommands.User
```

Output from these commands will be logged by the taskgraph runner in `/var/log/upstart/on-taskgraph.log`.

Workflow Progress Notification

RackHD workflow progress feature provides message notification mechanism to indicate status of an active workflow or task. User can get to know what has been done and what is to be done for an active workflow or task with progress messages.

Workflow Progress Events

RackHD will publish a workflow progress message if any of below events happens:

- Workflow started or finished events
- Task started or finished events
- RackHD marked important milestone events for an active long-run task.

In some cases RackHD can't easily get progress information, some milestones are created to divide a task into several small sections. Progress messages will be sent if any of those milestones is achieved.

- Progress timer timeout for an active long-run task.

Some tasks don't have milestones but progress information is continuous and can be got all the time. In this case progress messages is generated with fixed interval.

Progress Message Payload

4 attributes are used to describe progress information:

properties	Type	Description
maximum	Integer	Maximum step quantity for a workflow or a task. For tasks with continuous progress, it is 100.
value	Integer	Completed step quantity for a workflow or a task. For tasks with continuous progress, it varies from 0-100, which is inversely calculated from percentage and rounded to integer if calculation gives non-integer value.
percentage	String	Percentage of a workflow or task that is completed. Normally <i>value</i> divided by <i>maximum</i> will give <i>percentage</i> . However in the case that tasks have continuous progress, percentage is directly got. In this case <i>maximum</i> will be always set to 100 and <i>value</i> will be set to the percent number. For example, a percentage “65%” will give <i>maximum</i> 100 and <i>value</i> 65.
description	String	Short description for progress events

Below is an example of progress information payload for a workflow that has 4 steps and we have just finished the first step. Percentage is 25% given by 1 / 4.

```
progress: {
  value: 1,
  maximum: 4,
  description: 'Task "Install CentOS" started',
  percentage: '25%'
}
```

A complete RackHD progress message payload contains two levels of progress information (refer to [Workflow Progress Measurement](#)) as well as some useful information like graphId, graphName, nodeId, taskId and taskName, below is an example of a complete progress message:

```
{
  progress: {
    value: 1,
    maximum: 4,
    description: 'Task "Install CentOS" started',
    percentage: '25%'
  },
  graphName: 'Install CentOS',
  graphId: '12a8f275-7abf-46ee-834b-6aa34cce8d78',
  nodeId: '58542c752be86d0672cef383',
  taskProgress: {
    taskId: 'cb7d5793-abcfe4a7f-aef6-e768e999de1d',
    taskName: 'Install CentOS',
    progress: {
      value: 0,
      maximum: 4,
      description: 'Task started',
      percentage: '0%'
    }
  }
}
```

Though RackHD provides percentage number as progress measurement in progress message, most of the time workflow progress is based on events counting. RackHD progress message is not always proper to be used for workflow

executing time estimation.

Workflow Progress Measurement

RackHD progress information contains two levels of progress as shows in *Progress Message Payload Example* :

- *Task level progress*: progress measurement of the executing task of an active workflow.
- *Workflow level progress*: progress measurement of an active workflow.

Task progress is actually part of workflow progress. However task and workflow have two independent progress measurement methods.

Workflow level progress measurement Before a workflow's completion workflow level progress is based on tasks counting. It is measured by completed tasks count (which will be assigned to *value*) against total tasks count (which will be assigned to *maximum*) for the workflow.

Percentage will be set to 100% and *value* be set to *maximum* at workflow's completion. After completion workflow level progress will not be updated even though some tasks may still be running.

Task level progress measurement RackHD has different task level progress measurement methods for non-long-run tasks and two long-run tasks, OS installation tasks and secure erase task.

Non-long-run task progress

Each RackHD task has two progress events:

- *task started*
- *task finished*

A non-long-run task will complete in short time and only the started and finished events can be sensed. Thus only two progress messages will be published for non-long-run tasks.

Besides task started and finished events, a time-consuming task is not proper to only publish two events, thus different measurements are created.

OS installation task progress

As a typical long-run task, OS installation task progress can't be easily measured. As a compromise, RackHD creates some milestones at important timeslot of installation process thus divides OS install task into several sub-tasks.

Below table includes descriptions for all existing RackHD OS installation milestones:

Milestone name	Milestone description
requestProfile	Enter ipxe and request OS installation profile. Common milestone for all OSes.
enterProfile	Enter profile, start to download kernel or installer. Common milestone for all OSes.
startInstaller	Start installer and prepare installation. Common milestone for all OSes.
preConfig	Enter Pre OS configuration.
startSetup	Net use Windows Server 2012 and start setup.exe. Only used for Windows Server.
installToDisk	Execute OS installation. Only used for CoreOS.
startPartition	Start partition. Only used for Ubuntu.
postPartitioning	Finished partitioning and mounting, start package installation. Only used for SUSE.
chroot	Finished package installation, start first boot. Only used for SUSE.
postConfig	Enter Post OS configuration.
completed	Finished OS installation. Common milestone for all OSes.

Below table includes default milestone sequence for RackHD supported OSes:

OS Name	Milestone Quantity	Milestones in Sequence
CentOS, RHEL	6	1.requestProfile; 2.enterProfile; 3.startInstaller; 4.preConfig; 5.postConfig; 6.completed
Esxi	6	1.requestProfile; 2.enterProfile; 3.startInstaller; 4.preConfig; 5.postConfig; 6.completed
CoreOS	5	1.requestProfile; 2.enterProfile; 3.startInstaller; 4.installToDisk; 5.completed
Ubuntu	7	1.requestProfile; 2.enterProfile; 3.startInstaller; 4.preConfig; 5.startPartition; 6.postConfig; 7.completed
Win-dowServer	5	1.requestProfile; 2.enterProfile; 3.startInstaller; 4.startSetup; 5.completed
SUSE	7	1.requestProfile; 2.enterProfile; 3.startInstaller; 4.preConfig; 5.postPartitioning; 6.chroot; 7.completed
PhotonOS	5	1.requestProfile; 2.enterProfile; 3.startInstaller; 4.postConfig; 5.completed

In progress message, milestone quantity will be set to *maximum* and sequence number to *value* while RackHD is installing OS.

Secure erase task progress

For secure erase task, RackHD can get continuous percentage progress from node. Thus node is required to send the percentage data to RackHD with fixed interval. RackHD will receive and parse the percentage to get *value* and *maximum* and then publish progress message.

Progress Message Retrieve Channels

As instant data, progress messages can't be retrieved via API. Instead progress messages will be published in AMQP channel and posted to webhook urls after adding RackHD standard message header.

Below is basic information for user to retrieve data from AMQP channel:

- Exchange: on.events
- Routing Key: graph.progress.updated.information.<graphId>.<nodeId>

More details on RackHD AMQP events and webhook feature, please refer to [Event Notification](#).

Workflow Tasks

A workflow task is a unit of work decorated with data and logic that allows it to be included and run within a workflow. Tasks can be defined to do wide-ranging operations, such as bootstrap a server node into a Linux microkernel, parse data for matches against a rule, and others. The tasks in a workflow are run in a specific order.

A workflow task is made up of three parts:

- Task Definition
- Base Task Definition
- Job

Task Definitions

A task definition contains the basic description of the task. It contains the following fields.

Name	Type	Flags	Description
friendlyName	String	Required	A human-readable name for the task
injectableName	String	Required	A unique name used by the system and the API to refer to the task.
implementsTask	String	Required	The injectableName of the base task.
optionsSchema	Object/ String	Optional	The JSON schema for the task's <i>options</i> , see Options Schema for detail.
options	Object	Required	Key value pairs that are passed in as options to the job. Values required by a job may be defined in the task definition or overridden by options in a graph definition.
properties	Object	Required	JSON defining any relevant metadata or tagging for the task.

Below is a sample task definition in JSON for an Ubuntu installer.

```
{
  "friendlyName": "Install Ubuntu",
  "injectableName": "Task.Os.Install.Ubuntu",
  "implementsTask": "Task.Base.Os.Install",
  "options": {
    "username": "monorail",
    "password": "password",
    "profile": "install-trusty.ipxe",
    "hostname": "monorail",
    "uid": 1010,
    "domain": ""
  },
  "properties": {
    "os": {
      "linux": {
        "distribution": "ubuntu",
        "release": "trusty"
      }
    }
  }
}
```

Sample output (returns injectableName):

```
"Task.Os.Install.Ubuntu.Utopic"
```

Base Task Definitions

A Base Task Definition outlines validation requirements (an interface) and a common job to be used for a certain class of tasks. Base Task Definitions exist to provide strict and standardized validation schemas for graphs, and to improve code re-use and modularity.

The following table describes the fields of a Base Task Definition.

Name	Type	Flags	Description
friendly-Name	String	Re-quired	A human-readable name for the task.
injectable-Name	String	Re-quired	A unique name used by the system and the API to refer to the task.
optionsSchema	Object/ String	Optional	The JSON schema for the job's <i>options</i> , see Options Schema for detail.
requiredOptions	Object	Re-quired	Required option values to be set in a task definition implementing the base task.
required-Properties	Object	Re-quired	JSON defining required properties that need to exist in other tasks in a graph in order for this task to be able to be run successfully.
properties	Object	Re-quired	JSON defining any relevant metadata or tagging for the task. This metadata is merged with any properties defined in task definitions that implement the base task.

The following example shows the base task *Install Ubuntu* task definition:

```
{
  "friendlyName": "Install OS",
  "injectableName": "Task.Base.Os.Install",
  "runJob": "Job.Os.Install",
  "requiredOptions": [
    "profile"
  ],
  "requiredProperties": {
    "power.state": "reboot"
  },
  "properties": {
    "os": {
      "type": "install"
    }
  }
}
```

This base task is a generic Install OS task. It runs the job named *Job.Os.Install* and specifies that this job requires the option 'profile'. As a result, any task definition using the *Install OS* base task must provide at least these options to the OS installer job. These options are utilized by logic in the job.

```
this._subscribeRequestProfile(function() {
  return this.profile;
});
```

Another task definition that utilizes the above base task looks like:

```
{
  "friendlyName": "Install CoreOS",
  "injectableName": "Task.Os.Install.CoreOS",
  "implementsTask": "Task.Base.Os.Install",
  "options": {
    "username": "root",
    "password": "root",
    "profile": "install-coreos.ipxe",
    "hostname": "coreos-node"
  },
  "properties": {
    "os": {
```

```
        "linux": {
            "distribution": "coreos"
        }
    }
}
```

The primary difference between the *Install CoreOS* task and the *Install Ubuntu* task is the profile value, which is the ipxe template that specifies the installer images that an installation target should download.

Options Schema

The Options Schema is a [JSON-Schema](#) file or object that outlines the attributes and validation requirement for all options of a task or job. It provides standardized and declarative way to annotate task/job options. It offloads job's validation work and brings benefit to the upfront validation for graph input options.

Schema Classification

There are totally 3 kinds of options schema: Common options schema, Base Task options schema and Task options schema.

- The Common options schema is to describe all those common options that are shared by all tasks, such as `_taskTimeout`, the common options schema is defined in the file '<https://github.com/RackHD/on-tasks/blob/master/lib/task-data/schemas/common-task-options.json>'. User doesn't have to explicitly define the common schema in Task or Base Task definition, it is default enabled for every task.
- The schema in Base Task definition is to describe the options of the corresponding job.
- The schema in Task definition is to describe the options of corresponding task. Since a Task definition will always link to a Base Task, so the task's schema will automatically inherit the Base Task's schema during validation. So in practice, usually the task schema only needs to describe those options that are not covered in Base Task.

NOTE: The options schema is always optional for Task definition and Base Task definition. If options schema is not defined, that means user gives up the upfront options validation before running a TaskGraph.

Schema Format

The options schema supports two kinds of format:

- Built-in Schema <Object>: Directly put the full JSON schema content into the Task and Base Task definition.
- Schema File Reference <String>: Specify the file name of a JSON file, the JSON file is a valid JSON schema and it must be placed in the folder '<https://github.com/RackHD/on-tasks/tree/master/lib/task-data/schemas>'.

The Built-in Schema is usually used when there is few options or for situation that is not suitable to use file reference, such as within skupack. The File Reference schema is usually used when there are plents of options or to share schema between Task and Base Task.

Below is an example of Built-in Schema in Base Task definition:

```
{
  "friendlyName": "Analyze OS Repository",
  "injectableName": "Task.Base.Os.Analyze.Repo",
  "runJob": "Job.Os.Analyze.Repo",
  "optionsSchema": {
    "properties": {
      "version": {
        "$ref": "types-installos.json#/definitions/Version"
      },
      "repo": {
        "$ref": "types-installos.json#/definitions/Repo"
      }
    }
  }
}
```

```

    },
    "osName": {
      "enum": [
        "ESXi"
      ]
    },
    "required": [
      "osName",
      "repo",
      "version"
    ],
    "requiredProperties": {},
    "properties": {}
  }
}

```

Below is an example of File Reference schema in Base Task definition:

```

{
  "friendlyName": "Linux Commands",
  "injectableName": "Task.Base.Linux.Commands",
  "runJob": "Job.Linux.Commands",
  "optionsSchema": "linux-command.json",
  "requiredProperties": {},
  "properties": {
    "commands": {}
  }
}

```

Upfront Schema Validation

The options schema validation will be firstly executed when user triggers a workflow. Only if all options (Combine user input and the default value) conform to all of above schemas for the task, the workflow can then be successfully triggered. If any option violates the schema, The API request will report **400 Bad Request** and append detail error message in response body. For example:

Below is the message if user forgets the required option *version* while installing CentOS:

```
"message": "Task.Os.Install.CentOS: JSON schema validation failed - data should have required property 'version'"

```

Below is the message if the input *uid* beyond the allowed range.

```
"message": "Task.Os.Install.CentOS: JSON schema validation failed - data.users[0].uid should be >= 500"

```

Below is the message if the format of option *rootPassword* is not correct:

```
"message": "Task.Os.Install.CentOS: JSON schema validation failed - data.rootPassword should be string"

```

Task Jobs

A job is a javascript subclass with a run function that can be referenced by a string. When a new task is created, and all of its validation and setup logic handled, the remainder of its responsibility is to instantiate a new job class instance for its specified job (passing down the options provided in the definition to the job constructor) and run that job.

Defining a Job

To create a job, define a subclass of `Job.Base` that has a method called `_run` and calls `this._done()` somewhere, if the job is not one that runs indefinitely.

```
// Setup injector
module.exports = jobFactory;
di.annotate(jobFactory, new di.Provide('Job.example'));
di.annotate(jobFactory, new di.Inject('Job.Base');

// Dependency context
function jobFactory(BaseJob) {
  // Constructor
  function Job(options, context, taskId) {
    Job.super_.call(this, logger, options, context, taskId);
  }
  util.inherits(Job, BaseJob);

  // _run function called by base job
  Job.prototype._run = function _run() {
    var self = this;
    doWorkHere(args, function(err) {
      if (err) {
        self._done(err);
      } else {
        self._done();
      }
    });
  }

  return Job;
}
```

Many jobs are event-based by nature, so the base job provides many helpers for assigning callbacks to a myriad of AMQP events published by RackHD services, such as DHCP requests from a specific mac address, HTTP downloads from a specific IP, template rendering requests, etc.

Task Templates

There are some values that may be needed in a task definition which are not known in advance. In some cases, it is also more convenient to use placeholder values in a task definition than literal values. In these cases, a simple template rendering syntax can be used in task definitions. Rendering is also useful in places where two or more tasks need to use the same value (e.g. options.file), but it cannot be hardcoded ahead of time.

Task templates use [Mustache syntax](#), with some additional features detailed below. To define a value to be rendered, place it within curly braces in a string:

```
someOption: 'an option to be rendered: {{ options.renderedOption }}'
```

At render time, values are rendered if they exist in the task render context. The render context contains the following fields:

Field	Description
server	The server field contains all values found in the configuration for the on-taskgraph process (/opt/monorail/config.json) Example Usage: <code>{{ server.mongo.port }}</code>
api	<p>Values used for constructing API requests in a template:</p> <ul style="list-style-type: none"> • server – the base URI for the RackHD http server (e.g. <code>http://<server>:<port></code>) • base – the base http URI for the RackHD api (e.g. <code>http://<server>:<port>/api/current</code>) • templates – the base http URI for the RackHD api files route (e.g. <code>http://<server>:<port>/api/current/templates</code>) • profiles – the base http URI for the RackHD api files route (e.g. <code>http://<server>:<port>/api/current/profiles</code>) • lookups – the base http URI for the RackHD api files route (e.g. <code>http://<server>:<port>/api/current/lookups</code>) • files – the base http URI for the RackHD api files route (e.g. <code>http://<server>:<port>/api/current/files</code>) • nodes – the base http URI for the RackHD api nodes route (e.g. <code>http://<server>:<port>/api/current/nodes</code>)
file	<p>Values used for constructing static file server information in a template:</p> <ul style="list-style-type: none"> • server – the address of static file server (e.g. <code>http://<static-file-server>:<port></code>)
tasks	Allows access to instance variables of the task class instance created from the task definition. This is mainly used to access task.nodeId
options	This refers to the task definition options itself. Mainly for referencing values in substrings that will eventually be defined by a user (e.g. <code>'sudo mv {{ options.targetFile }} /tmp/{{ options.targetfile }}'</code>)
context	This refers to the shared context object that all tasks in a graph have R/W access to. Enables one task to use values produced by another at runtime. For example, the [ami catalog provider task](https://<server>:<port>/projects/RackHD/repos/on-tasks/browse/lib/task-data/tasks/provide-catalog-ami-bios-version.js) gets the most recent catalog entry for the AMI bios, whose value can be referenced by other tasks via <code>{{ context.ami.systemRomId }}</code>
sku	This refers to the SKU configuration data fetched from a SKU s. This field is added automatically if a SKU configuration exists in the the SKU s, rather than being specified by a user.
env	This refers to the environment configuration data retrieved from the environment database collection. Similar to sku, this field is added automatically rather than specified by a user.
1.6. RackHD Users Guide	

The download-files task is a good example of a task definition that makes use of multiple objects in the context:

```
{
  friendlyName: 'Flash MegaRAID Controller',
  injectableName: 'Task.Linux.Flash.LSI.MegaRAID',
  implementsTask: 'Task.Base.Linux.Commands',
  options: {
    file: null,
    downloadDir: '/opt/downloads',
    adapter: '0',
    commands: [
      'sudo /opt/MegaRAID/storcli/storcli64 /c{{ options.adapter }} download ' +
        'file={{ options.downloadDir }}/{{ options.file }} noverchk',
      'sudo /opt/MegaRAID/MegaCli/MegaCli64 -AdpSetProp -BatWarnDsbl 1 ' +
        '-a{{ options.adapter }}',
    ]
  },
  properties: {
    flash: {
      type: 'storage',
      vendor: {
        lsi: {
          controller: 'megaraid'
        }
      }
    }
  }
}
```

On creation, the options are rendered as below. The ‘file’ field is specified in this case by the contents of an API query, e.g. mr2208fw.rom

```
options: {
  file: 'mr2208fw.rom',
  downloadDir: '/opt/downloads',
  adapter: '0',
  commands: [
    'sudo /opt/MegaRAID/storcli/storcli64 /c0 download file=/opt/downloads/mr2208fw.rom noverchk',
    'sudo /opt/MegaRAID/MegaCli/MegaCli64 -AdpSetProp -BatWarnDsbl 1 -a0',
  ]
}
```

Task Rendering Features

For a full list of Mustache rendering features, including specifying conditionals and iterators, see the [Mustache man page](#)

Task templates also expand the capabilities of Mustache templating by adding the additional capabilities of *Fallback Rendering* and *Nested Rendering*, as documented below.

Fallback Rendering

Multiple values can be specified within the curly braces, separated by one or two ‘|’ characters (newlines are optional as well after the pipe character). In the case that the first value does not exist, the second one will be used, and so on. Values that are not prefixed by a context field (e.g. ‘options.’, ‘context.’) will be rendered as a plain string

```
// Unrendered
{
  <rest of task definition>
```



```

    options: {
      fallbackOption: 'this is a fallback option',
      value: '{{ options.doesNotExist || options.fallbackOption }}'
    }
  }
// Rendered
{
  <rest of task definition>
  options: {
    fallbackOption: 'this is a fallback option',
    value: 'this is a fallback option'
  }
}
// Unrendered, with fallback being a string
{
  <rest of task definition>
  options: {
    value: '{{ options.doesNotExist || fallbackString }}'
  }
}
// Rendered
{
  <rest of task definition>
  options: {
    value: 'fallbackString'
  }
}

```

Nested Rendering

Template rendering can go many levels deep. So if the rendered result of a template is itself another template, then rendering will continue until all values have been resolved, for example:

```

// Unrendered
{
  <rest of task definition>
  options: {
    value1: 'value1',
    value2: '{{ options.value1 }}',
    value3: 'a value: {{ options.value2 }}'
  }
}
// Rendered
{
  <rest of task definition>
  options: {
    value1: 'value1',
    value2: 'value1',
    value3: 'a value: value1'
  }
}

```

More examples

This task makes use of both template conditionals and iterators to generate a sequence of shell commands based on the options the task is created with.

```

{
  "friendlyName": "Delete RAID via Storcli",

```

```
"injectableName": "Task.Raid.Delete.MegaRAID",
"implementsTask": "Task.Base.Linux.Commands",
"options": {
  "deleteAll": true,
  "controller": 0,
  "raidIds": [], // [0,1,2]
  "path": "/opt/MegaRAID/storcli/storcli64",
  "commands": [
    "{{#options.deleteAll}}" +
    "sudo {{options.path}} /c{{options.controller}}/vall del force" +
    "{{/options.deleteAll}}" +
    "{{^options.deleteAll}}{{#options.raidIds}}" +
    "sudo {{options.path}} /c{{options.controller}}/v{{.}} del force;" +
    "{{/options.raidIds}}{{/options.deleteAll}}"
  ]
},
"properties": {}
}
```

If `options.deleteAll` is `true`, `options.commands` will be rendered as:

```
[
  "sudo /opt/MegaRAID/storcli/storcli64 /c0/vall del force"
]
```

If a user overrides `deleteAll` to be `false`, and `raidIds` to be `[0,1,2]`, then `options.commands` will become:

```
[
  "sudo /opt/MegaRAID/storcli/storcli64 /c0/v0 del force;sudo /opt/MegaRAID/storcli/storcli64 /c0/v1 del force;sudo /opt/MegaRAID/storcli/storcli64 /c0/v2 del force"
]
```

Task Timeouts

In the task options object, a magic value `_taskTimeout` can be used to specify a maximum amount of time a task may be run, in milliseconds. By default, this value is equal to 24 hours. To specify an infinite timeout, a value of 0 or -1 may be used.

```
{
  "options": {
    "_taskTimeout": 3600000 // 1 hour timeout (in ms)
  }
}
```

```
{
  "options": {
    "_taskTimeout": -1 // no timeout
  }
}
```

For backwards compatibility reasons, task timeouts can also be specified via the *`schedulerOverrides`* option:

```
{
  "options": {
    "schedulerOverrides": {
      "timeout": 3600000
    }
  }
}
```

If a task times out, it will cancel itself with a timeout error, and the task state in the database will equal “timeout”. The workflow engine will treat a task timeout as a failure and handle graph execution according to whether any other tasks handle a timeout exit value.

API Commands for Tasks

Get Available Tasks in the Library

```
GET /api/current/workflows/tasks/
```

```
curl <server>/api/current/workflows/tasks/
```

Create a Task Definition or a Base Task Definition

```
PUT /api/current/workflows/tasks
Content-Type: application/json
```

```
curl -X PUT \
-H 'Content-Type: application/json' \
-d <task definition>
<server>/api/current/workflows/tasks
```

Task Annotation

The RackHD Task Annotation is a schema for validating running tasks in the RackHD workflow engine, and is also used to provide self-hosted task documentation. Our build processes generate the files for this documentation.

Tasks that have been annotated have schema defined for them in the [on-tasks repository](#) under the directory [lib/task-data/schemas](#) using [JSON Schema](#)

How to Build Task Annotation Manually

```
git clone https://github.com/RackHD/on-http
cd on-http
npm install
npm run taskdoc
```

You can access it via **http(s)://<server>:<port>/taskdoc**, when on-http service is running.

For example:

TASK.OS.ESX.ANALYZE.REPO

TASK.OS.INSTALL.CENTOS

TASK.OS.INSTALL.COREOS

TASK.OS.INSTALL.ESXI

TASK.OS.INSTALL.PHOTONOS

TASK.OS.INSTALL.SUSE

TASK.OS.INSTALL.UBUNTU

TASK.OS.INSTALL.WIN

TASK.OS.RUN.EMC.DIAG

TASK.POLLERS.CREATEDEFAULT

TASK.PROCSHELLREBOOT

TASK.RAID.CREATE.MEGARAID

TASK.RAID.DELETE.MEGARAID

TASK.REDFISH.DISCOVERY

TASK.REMOVE.BMC.CREDENTIALS

TASK.RUN.UEFI

TASK.SET.BMC.CREDENTIALS

TASK.SFTP

TASK.SNMP.CATALOG

TASK.SNMP.COLLECT.DISCOVER

Task.Os.Esx.Analyze.Repo

Task.Os.Esx.Analyze.Repo

Analyze Esx Repository

```

{
  "friendlyName": "Analyze Esx Repository",
  "injectableName": "Task.Os.Esx.Analyze.Repo",
  "implementsTask": "Task.Base.Os.Analyze.Repo",
  "options": {
    "osName": "ESXi",
    "repo": "{{file.server}}/esxi/{{options.version}}"
  },
  "properties": {}
}

```

properties

Field	Type	Description
_taskTimeout <small>optional</small>	integer	This property can be used to specify a maximum amount of time a task may be run, in milliseconds. By default, this value is 24 hours. a value of 0 or -1 means infinite timeout minimum: <code>-1</code>
schedulerOverrides <small>optional</small>	object	properties: <code>timeout</code> See details for schedulerOverrides
version	string	The version of target OS pattern: <code>^[-0-9a-zA-Z_.*+]+</code>
repo	string	The OS http repository which contains all data that required for os installation pattern: <code>^http://</code> default: <code>{{file.server}}/esxi/{{options.version}}</code>
osName		value in: <ul style="list-style-type: none"> ESXi default: <code>ESXi</code>

Pollers

The pollers API provides functionality for periodic collection of IPMI and SNMP data.

IPMI

IPMI Pollers can be standalone or can be associated with a node. When an IPMI poller is associated with a node, it will attempt to use that node's IPMI OBM settings in order to communicate with the BMC. Otherwise, the poller must be manually configured with that node's IPMI settings.

If a node is found via discovery and contains a BMC catalog, then five IPMI pollers are automatically created for that node. The five pollers correspond to the “power”, “selInformation”, “sel”, “sdr” and “uid” (chassis LED) commands. These pollers do not collect data until the node has been configured with IPMI OBM settings.

Custom alerts for “sel” command IPMI pollers can be manually configured in their data definition, based on string and/or regex matching. IPMI pollers for the “sdr” command will automatically publish alerts onto an AMQP channel if any sensors of type “threshold” hold a value that does not equal “Not Available” or “ok”. See the Alerts section below for more information.

SNMP

SNMP pollers can be standalone or associated with a node. When an SNMP poller is associated with a node, it attempts to use that node's `snmpSettings` in order to communicate via SNMP. Otherwise, the poller must be manually configured with that node's SNMP settings.

If a node with "type": "switch" is created via the `/nodes` API with `autoDiscover` set to true, then six SNMP-based metric pollers will be created automatically for that node (see the Metric pollers section below for a list of these).

Example request to create and auto-discover a switch:

```
POST /api/current/nodes
Content-Type: application/json

{
  "name": "my switch",
  "identifiers": [],
  "ibms": [{"service": "snmp-ibm-service", "config": {"host": "10.1.1.3", "community": "public"}}],
  "type": "switch",
  "autoDiscover": true
}
```

Metric Pollers

In some cases, the data desired from a poller may require more complex processing than simply running an IPMI or SNMP command and parsing it. To address this, there is a poller type called a metric. A metric uses SNMP or IPMI, but can make multiples of these calls in aggregate and add post-processing logic to the results. There are currently six metrics available in the RackHD system:

- `snmp-interface-state`
- `snmp-interface-bandwidth-utilization`
- `snmp-memory-usage`
- `snmp-processor-load`
- `snmp-txrx-counters`
- `snmp-switch-sensor-status`

These metrics use SNMP to query multiple sources of information in order to calculate result data. For example, the bandwidth utilization metric calculates the delta between two sources of poll data at different times in order to produce data about how much network bandwidth is flowing through each interface.

API commands

When running the on-http process, these are some common API commands you can send:

Get available pollers in the library

```
GET /api/current/pollers/library
```

```
curl <server>/api/current/pollers/library
```

Create a new SNMP poller with a node

To use an SNMP poller that references a node, the node document must have an "ibms" field with a host and community fields:

```
// example node document with snmp settings
{
  "name": "example node",
  "identifiers": [],
  "ibms": [{ "service": "snmp-ibm-service", "config": { "host": "10.1.1.3", "community": "public" } }]
```

```
POST /api/current/pollers
{
  "type": "snmp",
  "pollInterval": 10000,
  "node": "54daadd764f1a8f1088fdc42",
  "config": {
    "oids": [
      "IF-MIB::ifSpeed",
      "IF-MIB::ifOperStatus"
    ]
  }
}
```

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"type":"snmp","pollInterval":10000,"node":"54daadd764f1a8f1088fdc42",
  "config":{"oids":["IF-MIB::ifSpeed","IF-MIB::ifOperStatus"]}}' \
<server>/api/current/pollers
```

Create a New IPMI Poller With a Node

```
POST /api/current/pollers
{
  "type": "ipmi",
  "pollInterval": 10000,
  "node": "54daadd764f1a8f1088fdc42",
  "config": {
    "command": "power"
  }
}
```

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"type":"ipmi","pollInterval":10000,"node":"54daadd764f1a8f1088fdc42",
  "config":{"command":"power"}}' \
<server>/api/current/pollers
```

```
{
  "node": "54daadd764f1a8f1088fdc42",
  "config": {
    "command": "power"
  },
  "pollInterval": 10000,
  "lastStarted": null,
  "lastFinished": null,
  "failureCount": 0,
  "createdAt": "2015-02-11T20:50:41.663Z",
  "updatedAt": "2015-02-11T20:50:41.663Z",
  "id": "54dbc0a11eaecfc22a30d59b",
  "type": "ipmi"
}
```

Create a New IPMI Poller Without a Node

```
POST /api/current/pollers
{
  "type": "ipmi",
  "pollInterval": 10000,
  "config": {
    "command": "power",
    "host": "10.1.1.2",
    "user": "admin",
    "password": "admin"
  }
}
```

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"type":"ipmi","pollInterval":10000,"node":"54daadd764f1a8f1088fdc42",
  "config":{"command":"power","host":"10.1.1.2","user":"admin","password":"admin"}}' \
<server>/api/current/pollers
```

```
{
  "node": null,
  "config": {
    "command": "power",
    "host": "10.1.1.2",
    "user": "admin",
    "password": "admin"
  },
  "pollInterval": 10000,
  "lastStarted": null,
  "lastFinished": null,
  "failureCount": 0,
  "createdAt": "2015-02-11T20:50:41.663Z",
  "updatedAt": "2015-02-11T20:50:41.663Z",
  "id": "54dbc0a11eaecfc22a30d59b",
  "type": "ipmi"
}
```

Create a New SNMP Poller

```
POST /api/current/pollers
{
  "type": "snmp",
  "pollInterval": 10000,
  "config": {
    "host": "10.1.1.3",
    "communityString": "public",
    "oids": [
      "PDU-MIB::outletVoltage",
      "PDU-MIB::outletCurrent"
    ]
  }
}
```

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"type":"snmp","pollInterval":10000,"node":"54daadd764f1a8f1088fdc42",
  "config":{"host":"10.1.1.3","communityString":"public",
    "oids":["PDU-MIB::outletVoltage","PDU-MIB::outletCurrent"]}}' \
```

```
<server>/api/current/pollers
```

```
{
  "node": null,
  "config": {
    "host": "10.1.1.3",
    "communityString": "public",
    "extensionMibs": [
      "PDU-MIB::outletVoltage",
      "PDU-MIB::outletCurrent"
    ]
  },
  "pollInterval": 10000,
  "lastStarted": null,
  "lastFinished": null,
  "failureCount": 0,
  "createdAt": "2015-02-11T20:50:41.663Z",
  "updatedAt": "2015-02-11T20:50:41.663Z",
  "id": "54dbc0a11eaecfc22a30d59b",
  "type": "snmp"
}
```

Create a New Metric Poller

Metric pollers can be created by adding the name of the metric to the poller config instead of data like “oids” or “command”

```
POST /api/current/pollers
{
  "type": "snmp",
  "pollInterval": 10000,
  "node": "54daadd764f1a8f1088fdc42",
  "config": {
    "metric": "snmp-interface-bandwidth-utilization"
  }
}
```

```
curl -X POST \
  -H 'Content-Type: application/json' \
  -d '{"type":"snmp","pollInterval":10000,"node":"54daadd764f1a8f1088fdc42",
    "config":{"metric":"snmp-interface-bandwidth-poller"}}' \
  <server>/api/current/pollers
```

Get a Poller’s Data Stream

```
GET /api/current/pollers/:id/data
```

```
curl <server>/api/current/pollers/<pollerid>/data
```

Sample Output: IPMI

```
[
  {
    "user": "admin",
    "password": "admin",
    "host": "10.1.1.2",
    "timestamp": "Wed Feb 11 2015 12:29:26 GMT-0800 (PST)",
    "sdr": [
      { "Lower critical": "0.000",
```



```

    "Upper critical": "87.000",
    "Sensor Id": "CPU1 Temp",
    "Normal Maximum": "89.000",
    "Lower non-critical": "0.000",
    "Status": "ok",
    "Entry Id Name": "Processor",
    "Upper non-critical": "84.000",
    "Sensor Type": "Temperature",
    "Entity Id": "3.1",
    "Nominal Reading": "45.000",
    "Sensor Reading": "31",
    "Sensor Reading Units": "degrees C",
    "Normal Minimum": "-4.000" },
  { "Lower critical": "0.000",
    "Upper critical": "87.000",
    "Sensor Id": "CPU2 Temp",
    "Normal Maximum": "89.000",
    "Lower non-critical": "0.000",
    "Status": "ok",
    "Entry Id Name": "Processor",
    "Upper non-critical": "84.000",
    "Sensor Type": "Temperature",
    "Entity Id": "3.2",
    "Nominal Reading": "45.000",
    "Sensor Reading": "25",
    "Sensor Reading Units": "degrees C",
    "Normal Minimum": "-4.000" },
  { "Lower critical": "-7.000",
    "Upper critical": "85.000",
    "Sensor Id": "System Temp",
    "Normal Maximum": "74.000",
    "Lower non-critical": "-5.000",
    "Status": "ok",
    "Entry Id Name": "System Board",
    "Upper non-critical": "80.000",
    "Sensor Type": "Temperature",
    "Entity Id": "7.1",
    "Nominal Reading": "45.000",
    "Sensor Reading": "30",
    "Sensor Reading Units": "degrees C",
    "Normal Minimum": "-4.000" },
  { "Lower critical": "-7.000",
    "Upper critical": "85.000",
    "Sensor Id": "Peripheral Temp",
    "Normal Maximum": "74.000",
    "Lower non-critical": "-5.000",
    "Status": "ok",
    "Entry Id Name": "System Board",
    "Upper non-critical": "80.000",
    "Sensor Type": "Temperature",
    "Entity Id": "7.2",
    "Nominal Reading": "45.000",
    "Sensor Reading": "41",
    "Sensor Reading Units": "degrees C",
    "Normal Minimum": "-4.000" },
  { "Lower critical": "-8.000",
    "Upper critical": "95.000",
    "Sensor Id": "PCH Temp",

```

```
"Normal Maximum": "67.000",
"Lower non-critical": "-5.000",
"Status": "ok",
"Entry Id Name": "System Board",
"Upper non-critical": "90.000",
"Sensor Type": "Temperature",
"Entity Id": "7.3",
"Nominal Reading": "45.000",
"Sensor Reading": "50",
"Sensor Reading Units": "degrees C",
"Normal Minimum": "-4.000" },
{ "Lower critical": "2.000",
  "Upper critical": "85.000",
  "Sensor Id": "P1-DIMMA1 TEMP",
  "Normal Maximum": "206.000",
  "Lower non-critical": "4.000",
  "Status": "ok",
  "Entry Id Name": "Memory Device",
  "Upper non-critical": "80.000",
  "Sensor Type": "Temperature",
  "Entity Id": "32.64",
  "Nominal Reading": "225.000",
  "Sensor Reading": "37",
  "Sensor Reading Units": "degrees C",
  "Normal Minimum": "168.000" },
{ "Lower critical": "2.000",
  "Upper critical": "85.000",
  "Sensor Id": "P1-DIMMB1 TEMP",
  "Normal Maximum": "206.000",
  "Lower non-critical": "4.000",
  "Status": "ok",
  "Entry Id Name": "Memory Device",
  "Upper non-critical": "80.000",
  "Sensor Type": "Temperature",
  "Entity Id": "32.65",
  "Nominal Reading": "225.000",
  "Sensor Reading": "37",
  "Sensor Reading Units": "degrees C",
  "Normal Minimum": "168.000" },
{ "Lower critical": "2.000",
  "Upper critical": "85.000",
  "Sensor Id": "P1-DIMMC1 TEMP",
  "Normal Maximum": "206.000",
  "Lower non-critical": "4.000",
  "Status": "ok",
  "Entry Id Name": "Memory Device",
  "Upper non-critical": "80.000",
  "Sensor Type": "Temperature",
  "Entity Id": "32.68",
  "Nominal Reading": "225.000",
  "Sensor Reading": "38",
  "Sensor Reading Units": "degrees C",
  "Normal Minimum": "168.000" },
{ "Lower critical": "2.000",
  "Upper critical": "85.000",
  "Sensor Id": "P1-DIMMD1 TEMP",
  "Normal Maximum": "206.000",
  "Lower non-critical": "4.000",
```

```

    "Status": "ok",
    "Entry Id Name": "Memory Device",
    "Upper non-critical": "80.000",
    "Sensor Type": "Temperature",
    "Entity Id": "32.69",
    "Nominal Reading": "225.000",
    "Sensor Reading": "38",
    "Sensor Reading Units": "degrees C",
    "Normal Minimum": "168.000" },
{
    "Lower critical": "2.000",
    "Upper critical": "85.000",
    "Sensor Id": "P2-DIMME1 TEMP",
    "Normal Maximum": "206.000",
    "Lower non-critical": "4.000",
    "Status": "ok",
    "Entry Id Name": "Memory Device",
    "Upper non-critical": "80.000",
    "Sensor Type": "Temperature",
    "Entity Id": "32.72",
    "Nominal Reading": "225.000",
    "Sensor Reading": "34",
    "Sensor Reading Units": "degrees C",
    "Normal Minimum": "168.000" },
{
    "Lower critical": "2.000",
    "Upper critical": "85.000",
    "Sensor Id": "P2-DIMMF1 TEMP",
    "Normal Maximum": "206.000",
    "Lower non-critical": "4.000",
    "Status": "ok",
    "Entry Id Name": "Memory Device",
    "Upper non-critical": "80.000",
    "Sensor Type": "Temperature",
    "Entity Id": "32.73",
    "Nominal Reading": "225.000",
    "Sensor Reading": "33",
    "Sensor Reading Units": "degrees C",
    "Normal Minimum": "168.000" },
{
    "Lower critical": "2.000",
    "Upper critical": "85.000",
    "Sensor Id": "P2-DIMMG1 TEMP",
    "Normal Maximum": "206.000",
    "Lower non-critical": "4.000",
    "Status": "ok",
    "Entry Id Name": "Memory Device",
    "Upper non-critical": "80.000",
    "Sensor Type": "Temperature",
    "Entity Id": "32.76",
    "Nominal Reading": "225.000",
    "Sensor Reading": "34",
    "Sensor Reading Units": "degrees C",
    "Normal Minimum": "168.000" },
{
    "Lower critical": "2.000",
    "Upper critical": "85.000",
    "Sensor Id": "P2-DIMMH1 TEMP",
    "Normal Maximum": "206.000",
    "Lower non-critical": "4.000",
    "Status": "ok",
    "Entry Id Name": "Memory Device",

```

```
"Upper non-critical": "80.000",
"Sensor Type": "Temperature",
"Entity Id": "32.77",
"Nominal Reading": "225.000",
"Sensor Reading": "34",
"Sensor Reading Units": "degrees C",
"Normal Minimum": "168.000" },
{ "Lower critical": "450.000",
  "Upper critical": "19050.000",
  "Sensor Id": "FAN1",
  "Normal Maximum": "12750.000",
  "Lower non-critical": "600.000",
  "Status": "ok",
  "Entry Id Name": "Fan Device",
  "Upper non-critical": "18975.000",
  "Sensor Type": "Fan",
  "Entity Id": "29.1",
  "Nominal Reading": "9600.000",
  "Sensor Reading": "4050",
  "Sensor Reading Units": "RPM",
  "Normal Minimum": "1500.000" },
{ "Lower critical": "450.000",
  "Upper critical": "19050.000",
  "Sensor Id": "FAN2",
  "Normal Maximum": "12750.000",
  "Lower non-critical": "600.000",
  "Status": "ok",
  "Entry Id Name": "Fan Device",
  "Upper non-critical": "18975.000",
  "Sensor Type": "Fan",
  "Entity Id": "29.2",
  "Nominal Reading": "9600.000",
  "Sensor Reading": "3975",
  "Sensor Reading Units": "RPM",
  "Normal Minimum": "1500.000" },
{ "Lower critical": "0.864",
  "Upper critical": "1.392",
  "Sensor Id": "VTT",
  "Normal Maximum": "1.648",
  "Lower non-critical": "0.912",
  "Status": "ok",
  "Entry Id Name": "System Board",
  "Upper non-critical": "1.344",
  "Sensor Type": "Voltage",
  "Entity Id": "7.10",
  "Nominal Reading": "1.488",
  "Sensor Reading": "1.008",
  "Sensor Reading Units": "Volts",
  "Normal Minimum": "1.344" },
{ "Lower critical": "0.512",
  "Upper critical": "1.520",
  "Sensor Id": "CPU1 Vcore",
  "Normal Maximum": "2.688",
  "Lower non-critical": "0.544",
  "Status": "ok",
  "Entry Id Name": "Processor",
  "Upper non-critical": "1.488",
  "Sensor Type": "Voltage",
```

```

    "Entity Id": "3.3",
    "Nominal Reading": "2.048",
    "Sensor Reading": "0.672",
    "Sensor Reading Units": "Volts",
    "Normal Minimum": "1.600" },
  { "Lower critical": "0.512",
    "Upper critical": "1.520",
    "Sensor Id": "CPU2 Vcore",
    "Normal Maximum": "2.688",
    "Lower non-critical": "0.544",
    "Status": "ok",
    "Entry Id Name": "Processor",
    "Upper non-critical": "1.488",
    "Sensor Type": "Voltage",
    "Entity Id": "3.4",
    "Nominal Reading": "2.048",
    "Sensor Reading": "0.688",
    "Sensor Reading Units": "Volts",
    "Normal Minimum": "1.664" },
  { "Lower critical": "1.152",
    "Upper critical": "1.696",
    "Sensor Id": "VDIMM ABCD",
    "Normal Maximum": "3.488",
    "Lower non-critical": "1.200",
    "Status": "ok",
    "Entry Id Name": "Memory Device",
    "Upper non-critical": "1.648",
    "Sensor Type": "Voltage",
    "Entity Id": "32.1",
    "Nominal Reading": "3.072",
    "Sensor Reading": "1.360",
    "Sensor Reading Units": "Volts",
    "Normal Minimum": "2.592" },
  { "Lower critical": "1.152",
    "Upper critical": "1.696",
    "Sensor Id": "VDIMM EFGH",
    "Normal Maximum": "3.488",
    "Lower non-critical": "1.200",
    "Status": "ok",
    "Entry Id Name": "Memory Device",
    "Upper non-critical": "1.648",
    "Sensor Type": "Voltage",
    "Entity Id": "32.2",
    "Nominal Reading": "3.072",
    "Sensor Reading": "1.344",
    "Sensor Reading Units": "Volts",
    "Normal Minimum": "2.592" },
  { "Lower critical": "0.928",
    "Upper critical": "1.264",
    "Sensor Id": "+1.1 V",
    "Normal Maximum": "2.416",
    "Lower non-critical": "0.976",
    "Status": "ok",
    "Entry Id Name": "System Board",
    "Upper non-critical": "1.216",
    "Sensor Type": "Voltage",
    "Entity Id": "7.11",
    "Nominal Reading": "2.192",

```

```
"Sensor Reading": "1.104",
"Sensor Reading Units": "Volts",
"Normal Minimum": "1.968" },
{ "Lower critical": "1.296",
  "Upper critical": "1.696",
  "Sensor Id": "+1.5 V",
  "Normal Maximum": "3.312",
  "Lower non-critical": "1.344",
  "Status": "ok",
  "Entry Id Name": "System Board",
  "Upper non-critical": "1.648",
  "Sensor Type": "Voltage",
  "Entity Id": "7.12",
  "Nominal Reading": "3.072",
  "Sensor Reading": "1.488",
  "Sensor Reading Units": "Volts",
  "Normal Minimum": "2.704" },
{ "Lower critical": "2.784",
  "Upper critical": "3.792",
  "Sensor Id": "3.3V",
  "Normal Maximum": "10.656",
  "Lower non-critical": "2.928",
  "Status": "ok",
  "Entry Id Name": "System Board",
  "Upper non-critical": "3.648",
  "Sensor Type": "Voltage",
  "Entity Id": "7.13",
  "Nominal Reading": "9.216",
  "Sensor Reading": "3.264",
  "Sensor Reading Units": "Volts",
  "Normal Minimum": "8.928" },
{ "Lower critical": "2.784",
  "Upper critical": "3.792",
  "Sensor Id": "+3.3VSB",
  "Normal Maximum": "7.296",
  "Lower non-critical": "2.928",
  "Status": "ok",
  "Entry Id Name": "System Board",
  "Upper non-critical": "3.648",
  "Sensor Type": "Voltage",
  "Entity Id": "7.14",
  "Nominal Reading": "6.624",
  "Sensor Reading": "3.312",
  "Sensor Reading Units": "Volts",
  "Normal Minimum": "5.952" },
{ "Lower critical": "4.288",
  "Upper critical": "5.696",
  "Sensor Id": "5V",
  "Normal Maximum": "10.560",
  "Lower non-critical": "4.480",
  "Status": "ok",
  "Entry Id Name": "System Board",
  "Upper non-critical": "5.504",
  "Sensor Type": "Voltage",
  "Entity Id": "7.15",
  "Nominal Reading": "10.112",
  "Sensor Reading": "4.928",
  "Sensor Reading Units": "Volts",
```

```

    "Normal Minimum": "9.280" },
  { "Lower critical": "4.288",
    "Upper critical": "5.696",
    "Sensor Id": "+5VSB",
    "Normal Maximum": "11.008",
    "Lower non-critical": "4.480",
    "Status": "ok",
    "Entry Id Name": "System Board",
    "Upper non-critical": "5.504",
    "Sensor Type": "Voltage",
    "Entity Id": "7.16",
    "Nominal Reading": "10.112",
    "Sensor Reading": "4.992",
    "Sensor Reading Units": "Volts",
    "Normal Minimum": "9.024" },
  { "Lower critical": "10.494",
    "Upper critical": "13.568",
    "Sensor Id": "12V",
    "Normal Maximum": "25.970",
    "Lower non-critical": "10.812",
    "Status": "ok",
    "Entry Id Name": "System Board",
    "Upper non-critical": "13.250",
    "Sensor Type": "Voltage",
    "Entity Id": "7.17",
    "Nominal Reading": "24.168",
    "Sensor Reading": "11.872",
    "Sensor Reading Units": "Volts",
    "Normal Minimum": "21.624" },
  { "Lower critical": "2.544",
    "Upper critical": "3.456",
    "Sensor Id": "VBAT",
    "Normal Maximum": "11.424",
    "Lower non-critical": "2.688",
    "Status": "ok",
    "Entry Id Name": "System Board",
    "Upper non-critical": "3.312",
    "Sensor Type": "Voltage",
    "Entity Id": "7.18",
    "Nominal Reading": "9.216",
    "Sensor Reading": "3.168",
    "Sensor Reading Units": "Volts",
    "Normal Minimum": "8.928" },
  { "Sensor Id": "PS1 Status",
    "Status": "ok",
    "States Asserted": "Presence detected",
    "Entity Id": "10.1" },
  { "Sensor Id": "PS2 Status",
    "Status": "ok",
    "States Asserted": "Presence detected",
    "Entity Id": "10.2" }
]
}
]

```

Sample Output: SNMP

```
[
{
  "host": "10.1.1.3",
  "communityString": "public",
  "extensionMibs": [
    "PDU-MIB::outletVoltage",
    "PDU-MIB::outletCurrent"
  ],
  "mibs": [
    [
      {
        "value": 116000,
        "name": "PDU-MIB::outletVoltage-1"
      },
      {
        "value": 116000,
        "name": "PDU-MIB::outletVoltage-2"
      },
      {
        "value": 116000,
        "name": "PDU-MIB::outletVoltage-3"
      },
      {
        "value": 116000,
        "name": "PDU-MIB::outletVoltage-4"
      },
      {
        "value": 116000,
        "name": "PDU-MIB::outletVoltage-5"
      },
      {
        "value": 117000,
        "name": "PDU-MIB::outletVoltage-6"
      },
      {
        "value": 117000,
        "name": "PDU-MIB::outletVoltage-7"
      },
      {
        "value": 117000,
        "name": "PDU-MIB::outletVoltage-8"
      }
    ],
    [
      {
        "value": 0,
        "name": "PDU-MIB::outletCurrent-1"
      },
      {
        "value": 0,
        "name": "PDU-MIB::outletCurrent-2"
      },
      {
        "value": 0,
        "name": "PDU-MIB::outletCurrent-3"
      },
      {
        "value": 0,
```



```

        "name": "PDU-MIB::outletCurrent-4"
    },
    {
        "value": 0,
        "name": "PDU-MIB::outletCurrent-5"
    },
    {
        "value": 737,
        "name": "PDU-MIB::outletCurrent-6"
    },
    {
        "value": 1538,
        "name": "PDU-MIB::outletCurrent-7"
    },
    {
        "value": 0,
        "name": "PDU-MIB::outletCurrent-8"
    }
]
],
"timestamp": "Wed Feb 11 2015 13:08:19 GMT-0800 (PST)"
},
{
    "host": "10.1.1.3",
    "communityString": "public",
    "extensionMibs": [
        "PDU-MIB::outletVoltage",
        "PDU-MIB::outletCurrent"
    ],
    "mibs": [
        [
            {
                "value": 117000,
                "name": "PDU-MIB::outletVoltage-1"
            },
            {
                "value": 117000,
                "name": "PDU-MIB::outletVoltage-2"
            },
            {
                "value": 117000,
                "name": "PDU-MIB::outletVoltage-3"
            },
            {
                "value": 117000,
                "name": "PDU-MIB::outletVoltage-4"
            },
            {
                "value": 117000,
                "name": "PDU-MIB::outletVoltage-5"
            },
            {
                "value": 117000,
                "name": "PDU-MIB::outletVoltage-6"
            },
            {
                "value": 117000,
                "name": "PDU-MIB::outletVoltage-7"
            }
        ]
    ]
}

```

```
    },
    {
      "value": 117000,
      "name": "PDU-MIB::outletVoltage-8"
    }
  ],
  [
    {
      "value": 0,
      "name": "PDU-MIB::outletCurrent-1"
    },
    {
      "value": 0,
      "name": "PDU-MIB::outletCurrent-2"
    },
    {
      "value": 0,
      "name": "PDU-MIB::outletCurrent-3"
    },
    {
      "value": 0,
      "name": "PDU-MIB::outletCurrent-4"
    },
    {
      "value": 0,
      "name": "PDU-MIB::outletCurrent-5"
    },
    {
      "value": 737,
      "name": "PDU-MIB::outletCurrent-6"
    },
    {
      "value": 1577,
      "name": "PDU-MIB::outletCurrent-7"
    },
    {
      "value": 0,
      "name": "PDU-MIB::outletCurrent-8"
    }
  ]
],
"timestamp": "Wed Feb 11 2015 13:08:25 GMT-0800 (PST)"
},
{
  "host": "10.1.1.3",
  "communityString": "public",
  "extensionMibs": [
    "PDU-MIB::outletVoltage",
    "PDU-MIB::outletCurrent"
  ],
  "mibs": [
    {
      "value": 116000,
      "name": "PDU-MIB::outletVoltage-1"
    },
    {
      "value": 116000,
```

```

    "name": "PDU-MIB::outletVoltage-2"
  },
  {
    "value": 116000,
    "name": "PDU-MIB::outletVoltage-3"
  },
  {
    "value": 116000,
    "name": "PDU-MIB::outletVoltage-4"
  },
  {
    "value": 116000,
    "name": "PDU-MIB::outletVoltage-5"
  },
  {
    "value": 117000,
    "name": "PDU-MIB::outletVoltage-6"
  },
  {
    "value": 117000,
    "name": "PDU-MIB::outletVoltage-7"
  },
  {
    "value": 117000,
    "name": "PDU-MIB::outletVoltage-8"
  }
],
[
  {
    "value": 0,
    "name": "PDU-MIB::outletCurrent-1"
  },
  {
    "value": 0,
    "name": "PDU-MIB::outletCurrent-2"
  },
  {
    "value": 0,
    "name": "PDU-MIB::outletCurrent-3"
  },
  {
    "value": 0,
    "name": "PDU-MIB::outletCurrent-4"
  },
  {
    "value": 0,
    "name": "PDU-MIB::outletCurrent-5"
  },
  {
    "value": 756,
    "name": "PDU-MIB::outletCurrent-6"
  },
  {
    "value": 1538,
    "name": "PDU-MIB::outletCurrent-7"
  },
  {
    "value": 0,

```

```
        "name": "PDU-MIB::outletCurrent-8"
      }
    ]
  },
  "timestamp": "Wed Feb 11 2015 13:08:30 GMT-0800 (PST)"
}
```

Get List of Active Pollers

```
GET /api/current/pollers
```

```
curl <server>/api/current/pollers
```

Get Definition for a Single Poller

```
GET /api/current/pollers/:id
```

```
curl <server>/api/current/pollers/<pollerid>
```

Update a Single Poller to change the interval

```
PATCH /api/current/pollers/:id
{
  "pollInterval": 15000
}
```

```
curl -X PATCH \
  -H 'Content-Type: application/json' \
  -d '{"pollInterval":15000}' \
  <server>/api/current/pollers/<pollerid>
```

Update a Single Poller to pause the poller

```
PATCH /api/current/pollers/:id
{
  "paused": true
}
```

```
curl -X PATCH \
  -H 'Content-Type: application/json' \
  -d '{"paused":true}' \
  <server>/api/current/pollers/<pollerid>
```

Delete a Single Poller

```
DELETE /api/current/pollers/:id
```

```
curl -X DELETE <server>/api/current/pollers/<pollerid>
```

Get List of Active Pollers Associated With a Node

```
GET /api/current/nodes/:id/pollers
```

```
curl <server>/api/current/nodes/<nodeid>/pollers
```

IPMI Poller Alerts

Please see [Event Notification](#) for more poller alert events information.

Sample data for a “sel” alert:

```
{
  "type": "polleralert",
  "action": "sel.updated",
  "typeId": "588586022116386a0d1e860f",
  "nodeId": "588585bee0f66f700da40335",
  "severity": "warning",
  "data": {
    "user": "admin",
    "host": "172.31.128.13",
    "alert": {
      "matches": [
        {
          "Event Type Code": "07",
          "Event Data": "/010000|040000/"
        }
      ],
      "reading": {
        "SEL Record ID": "0102",
        "Record Type": "02",
        "Timestamp": "01/01/1970 03:09:50",
        "Generator ID": "0001",
        "EvM Revision": "04",
        "Sensor Type": "Physical Security",
        "Sensor Number": "02",
        "Event Type": "Generic Discrete",
        "Event Direction": "Assertion Event",
        "Event Data": "010000",
        "Description": "Transition to Non-critical from OK",
        "Event Type Code": "07",
        "Sensor Type Code": "05"
      }
    }
  },
  "version": "1.0",
  "createdAt": "2017-01-23T07:36:53.092Z"
}
```

Sample data for an “sdr” alert:

```
{
  "type": "polleralert",
  "action": "sdr.updated",
  "typeId": "588586022116386a0d1e8610",
  "nodeId": "588585bee0f66f700da40335",
  "severity": "information",
  "data": {
    "host": "172.31.128.13",
    "user": "admin",
    "inCondition": true,
    "reading": {
      "sensorId": "Fan_SSD1 (0xfd)",
      "entityId": "29.1",
      "entryIdName": "Fan Device",
      "sdrType": "Threshold",
      "sensorType": "Fan",
      "sensorReading": "0",
      "sensorReadingUnits": "% RPM",
    }
  }
}
```

```
        "nominalReading": "",
        "normalMinimum": "",
        "normalMaximum": "",
        "statesAsserted": [],
        "status": "LowerCritical",
        "lowerCritical": "500.000",
        "lowerNonCritical": "1000.000",
        "positiveHysteresis": "Unspecified",
        "negativeHysteresis": "Unspecified",
        "minimumSensorRange": "Unspecified",
        "maximumSensorRange": "Unspecified",
        "eventMessageControl": "Per-threshold",
        "readableThresholds": "lcr lnc",
        "settableThresholds": "lcr lnc",
        "thresholdReadMask": "lcr lnc",
        "assertionsEnabled": ["lnc- lcr-"],
        "deassertionsEnabled": ["lnc- lcr-"]
    },
    "version": "1.0",
    "createdAt": "2017-01-23T07:36:56.179Z"
}
```

Sample data for an “snmp” alert:

```
{
  "type": "polleralert",
  "action": "snmp.updated",
  "typeId": "588586022116386a0d1e8611",
  "nodeId": "588585bee0f66f700da40335",
  "severity": "information",
  "data": {
    "states": {
      "last": "ON",
      "current": "OFF"
    }
  },
  data: {
    host: '10.1.1.3',
    oid: '.1.3.6.1.2.1.1.5.0',
    value: 'APC Rack Mounted UPS'
    matched: '/Mounted/'
  }
  "version": "1.0",
  "createdAt": "2017-01-23T08:20:32.231Z"
}
```

Sample data for an “snmp” metric alert:

```
{
  "type": "polleralert",
  "action": "snmp.updated",
  "typeId": "588586022116386a0d1e8611",
  "nodeId": "588585bee0f66f700da40335",
  "severity": "information",
  "data": {
    "states": {
      "last": "ON",
      "current": "OFF"
    }
  }
}
```

```

    }
  },
  data: {
    host: '127.0.0.1',
    oid: '.1.3.6.1.4.1.9.9.117.1.1.2.1.2.470',
    value: 'No Such Instance currently exists at this OID',
    matched: { contains: 'No Such Instance' },
    severity: 'warning',
    description: 'PSU element is not present',
    metric: 'snmp-switch-sensor-status'
  }
  "version": "1.0",
  "createdAt": "2017-01-23T08:20:32.231Z"
}

```

Creating Alerts

Alerting for sdr pollers is automatic and triggered when a threshold sensor has a value that does not equal either “ok” or “Not available”. In the example sdr alert above, the value being alerted is “nr”, for Non-recoverable.

Alerts for sel poller data are more flexible and can be user-defined via string or regex matching. The data structure for an sdr result has five keys: ‘date’, ‘time’, ‘sensor’, ‘event’ and ‘value’. Alert data can be specified via a JSON object that maps these keys to either exactly matched or regex matched values:

```

[
  {
    "sensor": "/Power Unit\s.*$/",
    "event": "Fully Redundant"
  }
]

```

In order for a value string to be interpreted as a regex pattern, it must begin and end with the ‘/’ character. Additionally, any regex escapes (e.g. n or s) must be double escaped before being serialized and sent over the wire (e.g. n becomes \n). In most programming languages, the equivalent of <RegexObject>.toString() will handle this serialization.

To add an alert to a poller, the above JSON schema must be added to the poller under config.alerts:

```

{
  "type": "ipmi",
  "pollInterval": 10000,
  "node": "54daadd764f1a8f1088fdc42",
  "config": {
    "command": "sel",
    "alerts": [
      {
        "sensor": "/Power Unit\s.*$/",
        "event": "Fully Redundant"
      },
      {
        "time": "/[0-3][0-3]:.*/",
        "sensor": "/Session Audit\s.*$/",
        "value": "Asserted"
      }
    ]
  }
}

```

Snmp poller alerts can be defined just like sel alerts via string or regex matching. However, the keys for an snmp alert must be a string or regex whose value you wish to check against the given OID numeric or string representation:

```
{
  "type": "snmp",
  "pollInterval": 10000,
  "node": "560ac7f33ab91d99448fb945",
  "config": {
    "alerts": [
      {
        ".1.3.6.1.2.1.1.5": "/Mounted/",
        ".1.3.6.1.2.1.1.1": "/ZA11/"
      }
    ],
    "oids": [
      ".1.3.6.1.2.1.1.1",
      ".1.3.6.1.2.1.1.5"
    ]
  }
}
```

Complex alerts are done by replacing the string/regex value with a validation object. The following example will match all OIDs with 'InErrors' in the name and generate an alert when the value is greater than 0.

```
{
  "type": "snmp",
  "pollInterval": 10000,
  "node": "560ac7f33ab91d99448fb945",
  "config": {
    "alerts": [
      {
        "/\\S*InErrors/": {
          "greaterThan": 0,
          "integer": true,
          "severity": "ignore"
        }
      }
    ],
    "metric": "snmp-txr-x-counters"
  }
}
```

Chassis Power State Alert

The IPMI chassis poller will publish an alert message when the power state of the node transitions. The AMQP message payload will contain both the current and last power state, a reference location to the node resource and a reference location to the pollers current data cache.

- Example message:

```
{
  "type": "polleralert",
  "action": "chassispower.updated",
  "typeId": "588586022116386a0dle8611",
  "nodeId": "588585bee0f66f700da40335",
  "severity": "information",
  "data": {
    "states": {
      "last": "ON",
      "current": "OFF"
    }
  }
}
```



```

    },
    "version": "1.0",
    "createdAt": "2017-01-23T08:20:32.231Z"
  }
}

```

Poller JSON Format

Pollers are defined via JSON with these required fields:

Name	Type	Flags	Description
type	String	required	Poller type. Valid values: ipmi , snmp
pollInterval	Number	required	Time in milliseconds to wait between polls.

The following fields are only valid for IPMI pollers:

Name	Type	Flags	Description
config	Object	required	Hash of configuration parameters.
config.command	String	required	IPMI command to run. Valid values: power , sel , sdr
config.host	String	<i>optional</i>	IP/Hostname of the node's BMC.
config.user	String	<i>optional</i>	IPMI username.
config.password	String	<i>optional</i>	IPMI password.
config.metric	String	<i>optional</i>	Run a metric poller instead of a simple IPMI query. Use instead of config.command .
node	String	<i>optional</i>	Node ID to associate this poller with dynamically look up IPMI settings.

The following fields are only valid for SNMP pollers:

Name	Type	Flags	Description
config	Object	required	Hash of configuration parameters.
config.host	String	<i>optional</i>	IP/Hostname of the node's BMC.
config.community	String	<i>optional</i>	SNMP community string.
config.oids	String[]	<i>optional</i>	Array of OIDs to poll.
config.metric	String	<i>optional</i>	Run a metric poller instead of a simple OID query. Use instead of config.oids .
node	String	<i>optional</i>	Node ID to associate this poller with dynamically look up SNMP settings.

The following fields can be PATCH'ed to change poller behavior:

Name	Type	Description
pollInterval	Number	Time in milliseconds to wait between polls.
paused	Boolean	Determines if the poller can be scheduled. Setting 'paused' to true will cause the poller to no longer be run when pollInterval expires

ARP Cache Poller

With the Address Resolution Protocol (ARP) cache poller service enabled, the RackHD lookup service will update MAC/IP bindings based on the Linux kernel's `/proc/net/arp` table. This ARP poller deprecates the need for running the DHCP lease file poller since any IP request made to the host will attempt to resolve the hardware addresses IP and update the kernel's ARP cache.

SKUs

The SKU API provides functionality to categorize nodes into groups based on data present in a node's catalogs. SKU matching is done using a series of rules. If all rules of a given SKU match the latest version of a node's catalog set, then that SKU will be assigned to the node.

Upon discovering a node, the SKU will be assigned based on all existing SKU definitions in the system. SKUs for all nodes will be re-generated whenever a SKU definition is added, updated or deleted.

A default graph can also be assigned to a SKU. When a node is discovered that matches the SKU, the specified graph will be executed on the node.

Example

With a node that has the following catalog fields:

```
{
  "source": "dmi",
  "data": {
    "Base Board Information": {
      "Manufacturer": "Intel Corporation"
    }
  },
  "memory": {
    "total": "32946864kB"
    "free": "31682528kB"
  }
  /* ... */
}
```

We could match against these fields with this SKU definition:

```
{
  "name": "Intel 32GB RAM",
  "rules": [
    {
      "path": "dmi.Base Board Information.Manufacturer",
      "contains": "Intel"
    },
    {
      "path": "dmi.memory.total",
      "equals": "32946864kB"
    }
  ]
}
```

In both cases, the “path” string starts with “dmi” to signify that the rule should apply to the catalog with a “source” value of “dmi”.

This example makes use of the “contains” and “equals” rules. See the table at the bottom of this document for a list of additional validation rules that can be applied.

Package Support (skupack)

The SKU package API provides functionality to override the set of files served to a node by on-http with SKU specific files. If a SKU requires additional operations during OS provisioning, the SKU package can be used to serve out SKU specific installation scripts that override the default scripts and perform those operations.

The SKU package can be upload to a specific SKU id or it can be bundled with a set of rules to register a SKU during the package upload.

API commands

When running the on-http process, these are some common API commands you can send.

Create a New SKU with a Node

```
POST /api/current/skus
{
  "name": "Intel 32GB RAM",
  "rules": [
    {
      "path": "dmi.Base Board Information.Manufacturer",
      "contains": "Intel"
    },
    {
      "path": "ohai.dmi.memory.total",
      "equals": "32946864kB"
    }
  ],
  "discoveryGraphName": "Graph.InstallCoreOS",
  "discoveryGraphOptions": {
    "username": "testuser",
    "password": "hello",
    "hostname": "mycoreos"
  }
}
```

```
{
  "name": "Intel 32GB RAM",
  "rules": [
    {
      "path": "dmi.dmi.base_board.manufacturer",
      "contains": "Intel"
    },
    {
      "path": "dmi.memory.total",
      "equals": "32946864kB"
    }
  ],
  "discoveryGraphName": "Graph.InstallCoreOS",
  "discoveryGraphOptions": {
    "username": "testuser",
    "password": "hello",
    "hostname": "mycoreos"
  },
  "createdAt": "2015-02-11T23:39:38.143Z",
  "updatedAt": "2015-02-11T23:39:38.143Z",
  "id": "54dbe83a380cc102b61e0f75"
}
```

Create a SKU to Auto-Configure IPMI Settings

```
POST /api/current/skus
{
  "name": "Default IPMI settings for Quanta servers",
  "discoveryGraphName": "Graph.Obm.Ipmi.CreateSettings",
  "discoveryGraphOptions": {
    "defaults": {
      "user": "admin",
      "password": "admin"
    }
  },
  "rules": [
    {
      "path": "bmc.IP Address"
    },
    {
      "path": "dmi.Base Board Information.Manufacturer",
      "equals": "Quanta"
    }
  ]
}
```

Get List of SKUs

```
GET /api/current/skus
```

```
curl <server>/api/current/skus
```

Get Definition for a Single SKU

```
GET /api/current/skus/:id
```

```
curl <server>/api/current/skus/<skuid>
```

Update a Single SKU

```
PATCH /api/current/skus/:id
{
  "name": "Custom SKU Name"
}
```

```
curl -X PATCH \
-H 'Content-Type: application/json' \
-d '{"name": "Custom SKU Name"}' \
<server>/api/current/skus/<skuid>
```

Delete a Single SKU

```
DELETE /api/current/skus/:id
```

```
curl -X DELETE <server>/api/current/skus/<skuid>
```

Register a new SKU with a pack

```
POST /api/current/skus/pack
```

```
curl -X POST --data-binary @pack.tar.gz <server>/api/current/skus/pack
```

Add a SKU pack

```
PUT /api/current/skus/:id/pack
```

```
curl -T pack.tar.gz <server>/api/current/skus/<skuid>/pack
```

Delete a SKU pack

```
DELETE /api/current/skus/:id/pack
```

```
curl -X DELETE <server>/api/current/skus/<skuid>/pack
```

SKU JSON format

SKUs are defined via JSON, with these required fields:

Name	Type	Flags	Description
name	String	required, unique	Unique name identifying this SKU definition.
rules	Object[]	required	Array of validation rules that define the SKU.
rules[].path	String	required	Path into the catalog to validate against.
rules[].equals	*	<i>optional</i>	Exact value to match against.
rules[].in	*[]	<i>optional</i>	Array of possibly valid values.
rules[].notIn	*[]	<i>optional</i>	Array of possibly invalid values.
rules[].contains	String	<i>optional</i>	A string that the value should contain.
rules[].notContains	String	<i>optional</i>	A string that the value should not contain.
rules[].greaterThan	Number	<i>optional</i>	Number that the value should be greater than.
rules[].lessThan	Number	<i>optional</i>	Number that the value should be less than.
rules[].min	Number	<i>optional</i>	Number that the value should be greater than or equal to.
rules[].max	Number	<i>optional</i>	Number that the value should be less than or equal to.
rules[].regex	String	<i>optional</i>	A regular expression that the value should match.
rules[].notRegex	String	<i>optional</i>	A regular expression that the value should not match.
discoveryGraphName	String	<i>optional</i>	Name of graph to run against matching nodes on discovery.
discoveryGraphOptions	Object	<i>optional</i>	Options to pass to the graph being run on node discovery.

SKU Pack tar.gz format

The SKU pack requires the ‘config.json’ to be at the root of the tar.gz file. A typical package may have static, template, profile, workflow and task directories.

```
tar tzf pack.tar.gz:
config.json
static/
static/common/
static/common/discovery.docker.tar.xz
templates/
templates/ansible.pub
templates/esx-ks
```

SKU Pack config.json format

```
{
  "name": "Intel 32GB RAM",
  "rules": [
    {
      "path": "dmi.Base Board Information.Manufacturer",
      "contains": "Intel"
    },
    {
      "path": "dmi.memory.total",
      "equals": "32946864kB"
    }
  ],
  "httpStaticRoot": "static",
  "httpTemplateRoot": "templates",
  "workflowRoot": "workflows",
  "taskRoot": "tasks",
  "httpProfileRoot": "profiles",
  "skuConfig" : {
    "key": "value",
    "key2" : {
      "key": "value"
    }
  }
}
```

Key	Description
httpStaticRoot	Contains static files to be served by on-http
httpTemplateRoot	Contains template files to be loaded into the templates library
workflowRoot	Contains graphs to be loaded into the workflow library
taskRoot	Contains tasks to be loaded into the tasks library
httpProfileRoot	Contains profile files to be loaded into the profiles library
skuConfig	Contains sku specific configuration to be loaded into the environment collection
version	(optional) Contains a version string for display use
description	(optional) Contains a description string for display use

Tags

The Tag API provides functionality to automatically categorize nodes into groups based on data present in a node's catalogs or by manually assigning a tag to a node. When done automatically, tag matching is done using a series of rules. If all rules of a given tag match the latest version of a node's catalog set, then that tag will be assigned to the node. A node may be assigned many tags, both automatically through rules matching or manually by the user.

Upon discovering a node, the tag will be assigned based on all existing tag definitions in the system. Tags for all nodes will be re-generated whenever a tag definition is added. Tags that are currently assigned to a node are not automatically removed from nodes when the rules backing a tag are deleted.

Example

With a node that has the following catalog fields:

```
{
  "source": "dmi",
  "data": {
    "Base Board Information": {
      "Manufacturer": "Intel Corporation"
    }
  }
}
```

```

    }
  },
  "memory": {
    "total": "32946864kB"
    "free": "31682528kB"
  }
  /* ... */
}

```

We could match against these fields with this tag definition:

```

{
  "name": "Intel 32GB RAM",
  "rules": [
    {
      "path": "dmi.Base Board Information.Manufacturer",
      "contains": "Intel"
    },
    {
      "path": "dmi.memory.total",
      "equals": "32946864kB"
    }
  ]
}

```

In both cases, the “path” string starts with “dmi” to signify that the rule should apply to the catalog with a “source” value of “dmi”.

This example makes use of the “contains” and “equals” rules. See the table at the bottom of this document for a list of additional validation rules that can be applied.

API commands

When running the on-http process, these are some common API commands you can send.

If you want to view or manipulate tags directly on nodes, please see the API notes at [node-api-tags-ref-label](#).

Create a New tag

```

POST /api/current/tags
{
  "name": "Intel-32GB-RAM",
  "rules": [
    {
      "path": "dmi.Base Board Information.Manufacturer",
      "contains": "Intel"
    },
    {
      "path": "ohai.dmi.memory.total",
      "equals": "32946864kB"
    }
  ]
}

```

Get List of tags

```

GET /api/current/tags

```

```
curl <server>/api/current/tags
```

Get Definition for a Single tag

```
GET /api/current/tags/:tagname
```

```
curl <server>/api/current/tags/<tagname>
```

Delete a Single tag

```
DELETE /api/current/tags/:tagname
```

```
curl -X DELETE <server>/api/current/tags/<tagname>
```

List nodes with a tag

```
GET /api/current/tags/:tagname/nodes
```

```
curl <server>/api/current/tags/<tagname>/nodes
```

Post a workflow to all nodes with a tag

```
POST /api/current/tags/:tagname/nodes/workflows
```

```
curl -H "Content-Type: application/json" -X POST -d @options.json <server>/api/current/tags/<tagname>
```

Tag JSON format

Tag objects are defined via JSON using these fields:

Name	Type	Flags	Description
name	String	required, unique	Unique name identifying this SKU definition.
rules	Object[]	required	Array of validation rules that define the SKU.
rules[].path	String	required	Path into the catalog to validate against.
rules[].equals	*	<i>optional</i>	Exact value to match against.
rules[].in	*[]	<i>optional</i>	Array of possibly valid values.
rules[].notIn	*[]	<i>optional</i>	Array of possibly invalid values.
rules[].contains	String	<i>optional</i>	A string that the value should contain.
rules[].notContains	String	<i>optional</i>	A string that the value should not contain.
rules[].greaterThan	Number	<i>optional</i>	Number that the value should be greater than.
rules[].lessThan	Number	<i>optional</i>	Number that the value should be less than.
rules[].min	Number	<i>optional</i>	Number that the value should be greater than or equal to.
rules[].max	Number	<i>optional</i>	Number that the value should be less than or equal to.
rules[].regex	String	<i>optional</i>	A regular expression that the value should match.
rules[].notRegex	String	<i>optional</i>	A regular expression that the value should not match.

Built-in Workflows

Discovery

Passive Hardware Discovery

Switch type nodes can be discovered either by running a discovery graph against them or creating via http calls with the autoDiscover field set to true.

Automatic Discovery A new node created by posting to `/api/current/node` will be automatically discovered if:

- the type is 'switch'
- it has an `ibms` field with the host to query and snmp community string
- the `autoDiscover` field is set to true

Create a Node to be Auto-Discovered

```
POST /api/current/nodes
{
  "name": "nodeName"
  "type": "switch",
  "autoDiscover": true
  "ibms": [{"service": "snmp-ibm-service", "config": {"host": "10.1.1.3", "community": "public"}}]
```

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"name":"nodeName", "type": "switch", "autoDiscover":true, \
  "ibms": [{"service": "snmp-ibm-service", "config": {"host": "10.1.1.3", "community": "public"}}]' \
<server>/api/current/nodes
```

```
{
  "type": "switch",
  "name": "nodeName",
  "autoDiscover": true,
  "service": "snmp-ibm-service",
  "config": {
    "host": "10.1.1.3"
  },
  "createdAt": "2015-07-27T22:03:45.353Z",
  "updatedAt": "2015-07-27T22:03:45.353Z",
  "id": "55b6aac1024fd1b349afc145"
}
```

Discover an existing device node If you want to discover a switch node manually either create the node without an `autoDiscover` option or set `autoDiscover` to false you can then run discovery against the node by posting to `/api/current/nodes/:identifier/workflows` and specifying the node id in the graph options, eg:

```
POST /api/current/nodes/55b6afba024fd1b349afc148/workflows
{
  "name": "Graph.Switch.Discovery",
  "options": {
    "defaults": {
      "nodeId": "55b6afba024fd1b349afc148"
    }
  }
}
```

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"name": "Graph.Switch.Discovery", \
  "options":{"defaults":{"nodeId": "55b6afba024fd1b349afc148"}}}' \
<server>/api/current/nodes/55b6afba024fd1b349afc148/workflows
```

You can also use this mechanism to discovery a compute server or PDU, simply using different settings. For example, a smart PDU:

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"name": "nodeName", "type": "pdu", \
  "ibms": [{"service": "snmp-ibm-service", "config": {"host": "10.1.1.3", "community": "public"}}] \
  <server>/api/current/nodes
```

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"name": "Graph.PDU.Discovery", \
  "options":{"defaults":{"nodeId": "55b6afba024fd1b349afc148"}}}' \
  <server>/api/1.1/nodes/55b6afba024fd1b349afc148/workflows
```

And a management server (or other server you do not want to or cannot to reboot to interrogate)

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"name": "nodeName", "type": "compute", \
  "obms": [ { "service": "ipmi-obm-service", "config": { "host": "10.1.1.3", \
  "user": "admin", "password": "admin" } } ] }' \
  <server>/api/current/nodes
```

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{"name": "Graph.MgmtSKU.Discovery", \
  "options":{"defaults":{"nodeId": "55b6afba024fd1b349afc148"}}}' \
  <server>/api/current/nodes/55b6afba024fd1b349afc148/workflows
```

Discovering/Configuring Network Switches

Utilizing network switch installation environments like POAP (Cisco), ZTP (Arista) and ONIE (Cumulus, etc.), RackHD offers the capability to discover, inventory, and configure network switches during bootstrap.

Active Discovery The terms “active discovery” and “passive discovery” are used by RackHD to differentiate between a discovery workflow that occurs as part of a switch bootup process, and may potentially make persistent changes to the switch operating system (active discovery), versus discovery workflow that queries out of band endpoints against an already-configured switch without making any persistent changes to it (e.g. SNMP polling).

During active discovery, by default the RackHD system will do light cataloging as part of the discovery process, generating enough data to identify the SKU/model of a switch in order to dynamically generate workflows and templates specific to it.

For example, active discovery of a Cisco switch booting with POAP (Power On Auto-Provisioning) will create a catalog document with source “version” that SKU definitions can be built against:

```
{
  "node" : ObjectId("5708438c3bfc361c5cca74dc"),
  "source" : "version",
  "data" : {
    "kern_uptm_secs" : "2",
    "kick_file_name" : "bootflash:///n3000-uk9-kickstart.6.0.2.U5.2.bin",
    "rr_service" : null,
    "loader_ver_str" : "N/A",
    "module_id" : "48x10GT + 6x40G Supervisor",
    "kick_tmstamp" : "03/17/2015 10:50:07",
    "isan_file_name" : "bootflash:///n3000-uk9.6.0.2.U5.2.bin",
    "sys_ver_str" : "6.0(2)U5(2)",
```

```

    "bootflash_size" : "2007040",
    "kickstart_ver_str" : "6.0(2)U5(2)",
    "kick_cmpl_time" : "3/17/2015 2:00:00",
    "chassis_id" : "Nexus 3172T Chassis",
    "proc_board_id" : "FOC1928169X",
    "memory" : "3793756",
    "kern_uptm_mins" : "6",
    "bios_ver_str" : "2.0.0",
    "cpu_name" : "Intel(R) Pentium(R) CPU @ 2.00GHz",
    "bios_cmpl_time" : "04/01/2014",
    "kern_uptm_hrs" : "0",
    "rr_usecs" : "981748",
    "isan_tmstamp" : "03/17/2015 12:29:49",
    "rr_sys_ver" : "6.0(2)U5(2)",
    "rr_reason" : "Reset Requested by CLI command reload",
    "rr_ctime" : "Fri Apr 8 23:35:28 2016",
    "header_str" : "Cisco Nexus Operating System (NX-OS) Software",
    "isan_cmpl_time" : "3/17/2015 2:00:00",
    "host_name" : "switch",
    "mem_type" : "kB",
    "kern_uptm_days" : "0",
    "power_seq_ver_str" : "Module 1: version v1.1"
  },
  "createdAt" : ISODate("2016-04-08T23:49:36.985Z"),
  "updatedAt" : ISODate("2016-04-08T23:49:36.985Z"),
  "_id" : ObjectId("57084390a2eb38385c3998b7")
}

```

Extending the Active Discovery Workflow RackHD utilizes the ability of most switch installation environments to run python scripts. This makes it easy to extend the active discovery process to produce custom catalogs, and deploy switch configurations and boot images.

It will be helpful to understand the RackHD concepts of a SKU and a Workflow before reading ahead.

SKU documentation: [SKUs](#)

Workflow documentation: [Workflow Graphs](#)

In order to extend the discovery process, a SKU definition must be created and added to the system (see [SKUs](#)). An example SKU definition that matches the above Cisco catalog might look like this:

```

{
  "name": "Cisco Nexus 3000 Switch - 54 port",
  "rules": [
    {
      "path": "version.chassis_id",
      "regex": "Nexus\\s\\d\\d\\d\\d\\d\\d\\w?\\sChassis"
    },
    {
      "path": "version.module_id",
      "equals": "48x10GT + 6x40G Supervisor"
    }
  ],
  "discoveryGraphName": "Graph.Switch.CiscoNexus3000.MyCustomWorkflow",
  "discoveryGraphOptions": {}
}

```

Using the `discoveryGraphName` field of the SKU definition, custom workflows can be triggered during switch installation. Creation of these workflows is detailed below.

For the examples below, let's start with an empty workflow definition for our custom switch workflow:

```
{
  "friendlyName": "My Custom Cisco Switch Workflow",
  "injectableName": "Graph.Switch.CiscoNexus3000.MyCustomWorkflow",
  "options": {},
  "tasks": []
}
```

Extending Cataloging

To collect custom catalog data from the switch, a Python script must be created for each catalog entry that can return either JSON or XML formatted data, and that is able to run on the target switch (all imported modules must exist, and the syntax must be compatible with the switch OS' version of Python).

Custom Python scripts must execute their logic within a single `main` function, that returns the catalog data, for example the following script catalogs SNMP group information on a Cisco Nexus switch:

1. Define a cataloging script

```
def main():
    import json
    # Python module names vary depending on nxos version
    try:
        from cli import clid
    except:
        from cisco import clid
    data = {}

    try:
        data['group'] = json.loads(clid('show snmp group'))
    except:
        pass

    return data
```

In this example, the cli module exists in the Nexus OS in order to run Cisco CLI commands.

2. Upload the script as a template

Next, the script must be uploaded as a template to the RackHD server:

```
# PUT https://<server>:<port>/api/current/templates/library/cisco-catalog-snmp-example.py
# via curl:
curl -X PUT -H "Content-type: text/raw" -d @<script path> https://<server>:<port>/api/current/templat
```

3. Add script to a workflow

Scripts are sent to the switch to be run via the Linux Commands task, utilizing the `downloadUrl` option. More information on this task can be found in the documentation for the `linux-commands-ref-label`

After adding the cataloging script as a template, add a task definition to the custom workflow, so now it becomes:

```
{
  "friendlyName": "My Custom Cisco Switch Workflow",
  "injectableName": "Graph.Switch.CiscoNexus3000.MyCustomWorkflow",
  "options": {},
  "tasks": [
    {
      "label": "catalog-switch-config",
      "taskDefinition": {
        "friendlyName": "Catalog Cisco Snmp Group",
```

```

        "injectableName": "Task.Inline.Catalog.Switch.Cisco.SnmpGroup",
        "implementsTask": "Task.Base.Linux.Commands",
        "options": {
            "commands": [
                {
                    "downloadUrl": "{{ api.templates }}/cisco-catalog-snmp-example.py?nodeId=",
                    "catalog": { "format": "json", "source": "snmp-group" }
                }
            ]
        },
        "properties": {}
    },
}
]
}

```

Deploying a startup config

In order to deploy a startup config to a switch, another Python script needs to be created that will download and copy the startup config, and a template must be created for the startup config file itself.

The below Python script deploys a startup config to a Cisco Nexus switch during POAP:

```

def main():
    # Python module names vary depending on nxos version
    try:
        from cli import cli
    except:
        from cisco import cli

    tmp_config_path = "volatile:poap.cfg"

    cli("copy <%=startupConfigUri%> %s vrf management" % tmp_config_path)
    cli("copy %s running-config" % tmp_config_path)
    cli("copy running-config startup-config")
    # copying to scheduled-config is necessary for POAP to exit on the next
    # reboot and apply the configuration
    cli("copy %s scheduled-config" % tmp_config_path)

```

The deploy script and startup config file should be uploaded via the templates API:

```

# Upload the deploy script
# PUT https://<server>:<port>/api/current/templates/library/deploy-cisco-startup-config.py
# via curl:
curl -X PUT -H "Content-type: text/raw" -d @<deploy script path> https://<server>:<port>/api/current/

# Upload the startup config
# PUT https://<server>:<port>/api/current/templates/library/cisco-example-startup-config
# via curl:
curl -X PUT -H "Content-type: text/raw" -d @<startup config path> https://<server>:<port>/api/current/

```

Note the ejb template variable used in the above python script (<%=startupConfigUri%>). This is used by the RackHD server to render its own API address dynamically, and must be specified within the workflow options.

Now the custom workflow can be updated again with a task to deploy the startup config:

```

{
    "friendlyName": "My Custom Cisco Switch Workflow",
    "injectableName": "Graph.Switch.CiscoNexus3000.MyCustomWorkflow",
    "options": {},
}

```

```

    "tasks": [
      {
        "label": "deploy-startup-config",
        "taskDefinition": {
          "friendlyName": "Deploy Cisco Startup Config",
          "injectableName": "Task.Inline.Switch.Cisco.DeployStartupConfig",
          "implementsTask": "Task.Base.Linux.Commands",
          "options": {
            "startupConfig": "cisco-example-startup-config",
            "startupConfigUri": "{{ api.templates }}/{{ options.startupConfig }}?nodeId={{ nodeId }}",
            "commands": [
              {
                "downloadUrl": "{{ api.templates }}/deploy-cisco-startup-config.py?nodeId={{ nodeId }}"
              }
            ]
          },
          "properties": {}
        },
      },
      {
        "label": "catalog-switch-config",
        "taskDefinition": {
          "friendlyName": "Catalog Cisco Snmp Group",
          "injectableName": "Task.Inline.Catalog.Switch.Cisco.SnmpGroup",
          "implementsTask": "Task.Base.Linux.Commands",
          "options": {
            "commands": [
              {
                "downloadUrl": "{{ api.templates }}/cisco-catalog-snmp-example.py?nodeId={{ nodeId }}",
                "catalog": { "format": "json", "source": "snmp-group" }
              }
            ]
          },
          "properties": {}
        },
      },
    ]
  }
}

```

Note that the `startupConfigUri` template variable is set in the options for the task definition, so that the deploy script can download the startup config from the right location.

In order to make this workflow more re-usable for a variety of switches, the `startupConfig` option can be specified as an override in the SKU definition using the `discoveryGraphOptions` field, for example:

```

{
  "name": "Cisco Nexus 3000 Switch - 24 port",
  "rules": [
    {
      "path": "version.chassis_id",
      "regex": "Nexus\\s\\d\\d\\d\\d\\d\\d\\w?\\sChassis"
    },
    {
      "path": "version.module_id",
      "equals": "24x10GT.*"
    }
  ],
  "discoveryGraphName": "Graph.Switch.CiscoNexus3000.MyCustomWorkflow",
  "discoveryGraphOptions": {

```

```

    "deploy-startup-config": {
        "startupConfig": "example-cisco-startup-config-24-port"
    }
}

```

Refresh Node Discovery

Compute type nodes can be re-discovered/refreshed either by running an immediate refresh discovery graph or a delayed refresh discovery graph using the same nodeID from the original discovery process. The node catalog(s) will be updated with new entries.

Immediate Refresh Node Discovery A node can be refreshed immediately by posting to /api/2.0/workflows with a payload. The node will be rebooted automatically and the node re-discovery process will start.

Immediate Node Re-discovery example

```

POST /api/2.0/workflows
{
  "name": "Graph.Refresh.Immediate.Discovery",
  "options": {
    "reset-at-start": {
      "nodeId": "<nodeId>"
    },
    "discovery-refresh-graph": {
      "graphOptions": {
        "target": "<nodeId>"
      },
      "nodeId": "<nodeId>"
    },
    "generate-sku": {
      "nodeId": "<nodeId>"
    },
    "generate-enclosure": {
      "nodeId": "<nodeId>"
    },
    "create-default-pollers": {
      "nodeId": "<nodeId>"
    },
    "run-sku-graph": {
      "nodeId": "<nodeId>"
    },
    "nodeId": "<nodeId>"
  }
}

```

```

curl -X POST \
-H 'Content-Type: application/json' \
-d '{ "name": "Graph.Refresh.Immediate.Discovery",
      "options": {
        "reset-at-start": {
          "nodeId": "<nodeId>"
        },
        "discovery-refresh-graph": {
          "graphOptions": {
            "target": "<nodeId>"
          }
        }
      }
}'

```

```
        },
        "nodeId": "<nodeId>"
    },
    "generate-sku": {
        "nodeId": "<nodeId>"
    },
    "generate-enclosure": {
        "nodeId": "<nodeId>"
    },
    "create-default-pollers": {
        "nodeId": "<nodeId>"
    },
    "run-sku-graph": {
        "nodeId": "<nodeId>"
    },
    "nodeId": "<nodeId>"
}
}' \
<server>/api/2.0/workflows
```

Delayed Refresh Node Discovery A user can defer a node discovery by posting to `/api/2.0/workflows` with a payload. The user will need to manually reboot the node after executing the API before the node re-discovery/refresh process can start.

Delayed Node Re-discovery example

```
POST /api/2.0/workflows
{
  "name": "Graph.Refresh.Delayed.Discovery",
  "options": {
    "discovery-refresh-graph": {
      "graphOptions": {
        "target": "<nodeId>"
      },
      "nodeId": "<nodeId>"
    },
    "generate-sku": {
      "nodeId": "<nodeId>"
    },
    "generate-enclosure": {
      "nodeId": "<nodeId>"
    },
    "create-default-pollers": {
      "nodeId": "<nodeId>"
    },
    "run-sku-graph": {
      "nodeId": "<nodeId>"
    },
    "nodeId": "<nodeId>"
  }
}
```

```
curl -X POST \
-H 'Content-Type: application/json' \
-d '{ "name": "Graph.Refresh.Delayed.Discovery",
      "options": {
        "discovery-refresh-graph": {
```



```

        "graphOptions": {
            "target": "<nodeId>"
        },
        "nodeId": "<nodeId>"
    },
    "generate-sku": {
        "nodeId": "<nodeId>"
    },
    "generate-enclosure": {
        "nodeId": "<nodeId>"
    },
    "create-default-pollers": {
        "nodeId": "<nodeId>"
    },
    "run-sku-graph": {
        "nodeId": "<nodeId>"
    },
    "nodeId": "<nodeId>"
}
}' \
<server>/api/2.0/workflows

```

Manually rebooting the node using ipmitool example

```
ipmitool -H <BMC host IP address> -U <username> -P <password> chassis power reset
```

OS Installation

RackHD workflow support installing Operating System automatically from remote http repository.

Setting up RackHD OS repository with image service

Image servie provides a convinient method for users to setup OS repository, details please refer to <https://github.com/RackHD/image-service>. Alternatively, users could setup the repos by executing *Configuring RackHD OS Mirrors* and *Making the Mirrors*.

Configuring RackHD OS Mirrors

Setting up a Windows OS repo

- **Mounting the OS Image:**

Windows' installation requires that Windows OS' ISO image must be mounted to a directory accessable to the node. In the example below a windows server 2012 ISO image is being mounted to a directory name **Licensedwin2012**

```
sudo mount -o loop /var/renasar/on-http/static/http/W2K2012_2015-06-08_1040.iso /var/renasar/on-http/
```

- **Export the directory**

Edit the samba config file in order to export the shared directory

```
sudo nano /etc/samba/smb.conf
```

```

[windowsServer2012]
    comment = not windows server 201
    path = /var/renasar/on-http/static/http/Licensedwin2012

```

```
browseable = yes
guest ok = yes
writable = no
printable = no
```

- **Restart the samba share**

```
sudo service samba restart
```

MIRRORS

Mirrors may not be something you need to configure if you're utilizing the proxy capabilities of RackHD. If you previously configured proxies to support OS installation workflows, then those should be configured to provide access to the files needed. If you do not, or can not, utilize proxies, then you can host the OS installation files directly on the same instance with RackHD. The following instructions show how to create OS installation mirrors in support of the existing workflows.

Set the Links to the Mirrors:

```
sudo ln -s /var/mirrors/ubuntu <on-http directory>/static/http/ubuntu
sudo ln -s /var/mirrors/ubuntu/14.04/mirror/mirror.pnl.gov/ubuntu/ <on-http directory>/static/http/ubuntu/14.04/mirror/mirror.pnl.gov/ubuntu/
sudo ln -s /var/mirrors/centos <on-http directory>/static/http/centos
sudo ln -s /var/mirrors/suse <on-http directory>/static/http/suse
```

Making the Mirrors

Centos 6.5

Notes: Because CentOS 6.5 was deprecated by provider, the following rsync source is not available to make mirror. Just leave it for an example.

```
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" \
--exclude "i386" rsync://centos.eecs.wsu.edu/centos/6.5/ /var/mirrors/centos/6.5
```

Centos 7.0

```
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" \
--exclude "i386" rsync://centos.eecs.wsu.edu/centos/7/ /var/mirrors/centos/7
```

OpenSuse 12.3

```
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" --exclude "i386" --exclude "x86_64" \
rsync://opensuse.eecs.wsu.edu/opensuse/12.3/ /var/mirrors/opensuse/12.3
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" --exclude "i386" --exclude "x86_64" \
rsync://opensuse.eecs.wsu.edu/opensuse/12.3/ /var/mirrors/opensuse/12.3
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" --exclude "i386" --exclude "x86_64" \
rsync://opensuse.eecs.wsu.edu/opensuse/12.3/ /var/mirrors/opensuse/12.3
```

OpenSuse 13.1

```
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" --exclude "i386" --exclude "x86_64" \
rsync://opensuse.eecs.wsu.edu/opensuse/13.1/ /var/mirrors/opensuse/13.1
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" --exclude "i386" --exclude "x86_64" \
rsync://opensuse.eecs.wsu.edu/opensuse/13.1/ /var/mirrors/opensuse/13.1
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" --exclude "i386" --exclude "x86_64" \
rsync://opensuse.eecs.wsu.edu/opensuse/13.1/ /var/mirrors/opensuse/13.1
```

OpenSuse 13.2

```
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" --exclude "i386" --exclude "x86_64" \
rsync://opensuse.eecs.wsu.edu/opensuse/13.2/ /var/mirrors/opensuse/13.2
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" --exclude "i386" --exclude "x86_64" \
rsync://opensuse.eecs.wsu.edu/opensuse/13.2/ /var/mirrors/opensuse/13.2
sudo rsync --progress -av --delete --delete-excluded --exclude "local*" --exclude "i386" --exclude "x86_64" \
rsync://opensuse.eecs.wsu.edu/opensuse/13.2/ /var/mirrors/opensuse/13.2
```

Ubuntu

[IMPORTANT] DNS server is required in Ubuntu installation, make sure you have put following lines in `/etc/dhcp/dhcpd.conf`. 172.31.128.1 is a default option in RackHD

```
option domain-name-servers 172.31.128.1;
option routers 172.31.128.254;
```

[Method-1] For iso installation, see this payload https://github.com/RackHD/RackHD/blob/master/example/samples/install_ubuntu_payload_iso_minimal.json. Remember to replace `{{ file.server }}` with your own, see 'fileServerAddress' and 'fileServerPort' in `/opt/monorail/config.json`

```
mkdir ~/iso && cd ~/iso

# Download iso file
wget http://releases.ubuntu.com/13.04/ubuntu-14.04.5-server-amd64.iso

# Create mirror folder
mkdir -p /var/mirrors/ubuntu

# Replace {on-http-dir} with your own
mkdir -p {on-http-dir}/static/http/mirrors

# Mount iso
sudo mount ubuntu-14.04.5-server-amd64.iso /var/mirrors/ubuntu

sudo ln -s /var/mirrors/ubuntu {on-http-dir}/static/http/mirrors/

# Create workflow
# Replace the 9090 port if you are using other ports
# You can configure the port in /opt/monorail/config.json -> 'httpEndpoints' -> 'northbound-api-
curl -X POST -H 'Content-Type: application/json' -d @install_ubuntu_payload_iso_minimal.json 127.0.0.1
```

[Method-2] For live installation, see this payload https://github.com/RackHD/RackHD/blob/master/example/samples/install_ubuntu_payload_live_minimal.json. Remember to replace **repo** with your own `{fileServerAddress}:{fileServerPort}/ubuntu`, you can find the proper parameters in `/opt/monorail/config.json`

Add following block into `httpProxies` in `/opt/monorail/config.json`

```
{
  "localPath": "/ubuntu",
  "server": "http://us.archive.ubuntu.com/",
  "remotePath": "/ubuntu/"
}
```

```
# Create workflow
# Replace the 9090 port if you are using other ports
# You can configure the port in /opt/monorail/config.json -> 'httpEndpoints' -> 'northbound-api-
curl -X POST -H 'Content-Type: application/json' -d @install_ubuntu_payload_minimal.json 127.0.0.1
```

[Method-3] For the **Ubuntu repo**, you need some additional installation. The mirrors are easily made on Ubuntu, but not so easily replicated on other OS. On any recent distribution of Ubuntu:

```
# make the mirror directory (can sometimes hit a permissions issue)
sudo mkdir -p /var/mirrors/ubuntu/14.04/mirror
# create a file in /etc/apt/mirror.list (config below)
sudo vi /etc/apt/mirror.list
# run the mirror
sudo apt-mirror
```

```
##### config #####
#
set base_path    /var/mirrors/ubuntu/14.04
#
# set mirror_path  $base_path/mirror
# set skel_path    $base_path/skel
# set var_path     $base_path/var
# set cleanscript $var_path/clean.sh
# set defaultarch  <running host architecture>
# set postmirror_script $var_path/postmirror.sh
# set run_postmirror 0
set nthreads     20
set _tilde 0
#
##### end config #####

deb-amd64 http://mirror.pnl.gov/ubuntu trusty main
deb-amd64 http://mirror.pnl.gov/ubuntu trusty-updates main
deb-amd64 http://mirror.pnl.gov/ubuntu trusty-security main
clean http://mirror.pnl.gov/ubuntu

#end of file
#####
```

Debian For live installation, see this payload https://github.com/RackHD/RackHD/blob/master/example/samples/install_debian_payload_minimal.json. Remember to replace **repo** with your own **{fileServerAddress}:{fileServerPort}/debian**, you can find the proper parameters in `/opt/monorail/config.json`

Add following block into `httpProxies` in `/opt/monorail/config.json`

```
{
  "localPath": "/debian",
  "server": "http://ftp.us.debian.org/",
  "remotePath": "/debian/"
}
```

```
# Create workflow
# Replace the 9090 port if you are using other ports
# You can configure the port in /opt/monorail/config.json -> 'httpEndpoints' -> 'northbound-api'
curl -X POST -H 'Content-Type: application/json' -d @install_debian_payload_minimal.json 127.0.0.1:9090
```

Reference: Here are some useful links to vendor's official website about Mirros setup.

- **CentOS:** [Creating Local Mirrors for Updates or Installs](#)

Supported OS Installation Workflows

Supported OSes and their workflows are listed in table, and the listed versions have been verified by RackHD, but not limited to these, this table will be updated when more versions are verified.

OS	Workflow	Version
ESXi	Graph.InstallESXi	5.5/6.0
RHEL	Graph.InstallRHEL	7.0/7.1/7.2
CentOS	Graph.InstallCentOS	6.5/7
Ubuntu	Graph.InstallUbuntu	trusty(14.04)/xenial(16.04)/artful(17.10)
Debian	Graph.InstallDebian	wheezy(7)/jessie(8)/stretch(9)
SUSE	Graph.InstallSUSE	openSUSE: leap/42.1, SLES: 11/12
CoreOS	Graph.InstallCoreOS	899.17.0
Windows	Graph.InstallWindowsServer	Server 2012
PhotonOS	Graph.InstallPhotonOS	1.0

OS Installation Workflow APIs

An example of starting an OS installation workflow for CentOS:

```
curl -X POST \
  -H 'Content-Type: application/json' \
  -d @params.json \
  <server>/api/current/nodes/<identifier>/workflows?name=Graph.InstallCentOS
```

An example of params.json with minimal parameters for installing CentOS workflow:

```
{
  "options": {
    "defaults": {
      "version": "7",
      "repo": "http://172.31.128.1:9080/mirrors/centos/7/os/x86_64"
    }
  }
}
```

An example of params.json with full set of parameters for installing CentOS workflow:

```
{
  "options": {
    "defaults": {
      "version": "7",
      "repo": "http://172.31.128.1:9080/mirrors/centos/7/os/x86_64",
      "rootPassword": "root12345",
      "hostname": "rackhd-node",
      "domain": "abc.com",
      "users": [
        {
          "name": "rackhd1",
          "password": "123456",
          "uid": 1010,
          "sshKey": "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDJQ631/sw3D40h/6JfA+PFVy5Ofz6eu7caxl"
        },
        {
          "name": "rackhd2",
          "password": "123456",
          "uid": 1011,
          "sshKey": "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDJQ631/sw3D40h/6JfA+PFVy5Ofz6eu7caxl"
        }
      ],
      "rootSshKey": "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDJQ631/sw3D40h/6JfA+PFVy5Ofz6eu7caxl",
      "dnsServers": ["172.12.88.91", "192.168.20.77"],
    }
  }
}
```

```
"networkDevices": [
  {
    "device": "eth0",
    "ipv4": {
      "ipAddr": "192.168.1.29",
      "gateway": "192.168.1.1",
      "netmask": "255.255.255.0",
      "vlanIds": [104, 105]
    },
    "ipv6": {
      "ipAddr": "fec0::6ab4:0:5efe:157.60.14.21",
      "gateway": "fe80::5efe:131.107.25.1",
      "netmask": "ffff.ffff.ffff.ffff.0.0.0.0",
      "vlanIds": [101, 106]
    }
  },
  {
    "device": "eth1",
    "ipv4": {
      "ipAddr": "192.168.11.89",
      "gateway": "192.168.11.1",
      "netmask": "255.255.255.0",
      "vlanIds": [105, 109]
    },
    "ipv6": {
      "ipAddr": "fec0::6ab4:0:5efe:159.45.14.11",
      "gateway": "fe80::5efe:131.107.25.100",
      "netmask": "ffff.ffff.ffff.ffff.0.0.0.0",
      "vlanIds": [106, 108]
    }
  }
],
"kvm": true,
"installDisk": "/dev/sda",
"installPartitions": [
  {
    "mountPoint": "/boot",
    "size": "500",
    "fsType": "ext3"
  },
  {
    "mountPoint": "swap",
    "size": "500",
    "fsType": "swap"
  },
  {
    "mountPoint": "/",
    "size": "auto",
    "fsType": "ext3"
  }
]
}
```

There are few more payload examples at <https://github.com/RackHD/RackHD/tree/master/example/samples>.

Check the active workflow running on a node

```
curl <server>/api/current/nodes/<identifier>/workflows?active=true
```

Deprecated 1.1 API - Check the active workflow running on a node

```
curl <server>/api/1.1/nodes/<identifier>/workflows/active
```

Stop the currently active workflow on a node:

```
curl -X PUT \
-H 'Content-Type: application/json' \
-d '{"command": "cancel"}' \
<server>/api/current/nodes/<id>/workflows/action
```

Deprecated 1.1 API - Stop the active workflow to cancel OS installation:

```
curl -X DELETE <server>/api/1.1/nodes/<identifier>/workflows/active
```

Non-Windows OS Installation Workflow Payload

All parameters descriptions of OS installation workflow payload are listed below, they are fit for use with all supported OSes except for CoreOS (see note below).

NOTE: The CoreOS installer is pretty basic, and only supports certain parameters shown below. Configurations not directly supported by RackHD may still be made via a custom Ignition template. Typical parameters for CoreOS include: *version*, *repo*, and *installScriptUri* | **ignitionScriptUri* and optionally *vaultToken* and *grubLinuxAppend*.

Parameters	Type	Flags	Description
version	String	required	The version number of target OS that needs to install. NOTE: For Ubuntu, <i>version</i> should be the codename, not numbers, for example, it should be “trusty”, not “14.04”
repo	String	required	The OS repository address, currently only supports HTTP. Some examples of free OS distributions for reference. For CentOS , http://mirror.centos.org/centos/7/os/x86_64/ . For Ubuntu , http://us.archive.ubuntu.com/ubuntu/ . For openSUSE , http://download.opensuse.org/distribution/leap/42.1/repo/oss/ . For ESXi , RHEL , SLES and PhotonOS , the repository is the directory of mounted DVD ISO image, and http service is provided for this directory.
osName	String	required	(Debian/Ubuntu only) The OS name, the default value is debian for ubuntu installation use ubuntu .
rootPassword	String	<i>optional</i>	The password for the OS root account, it could be clear text, RackHD will do encryption before store it into OS installer’s config file. default <i>rootPassword</i> is “ RackHDRocks! ”. Some OS distributions’ password requirements <i>must</i> be satisfied. For ESXi 5.5 , ESXi 5 Password Requirements . For ESXi 6.0 , ESXi 6 Password Requirements .
hostname	String	<i>optional</i>	The hostname for target OS, default <i>hostname</i> is “ localhost ”
domain	String	<i>optional</i>	The domain for target OS
timezone	String	<i>optional</i>	(Debian/Ubuntu only) The Timezone based on \$TZ. Please refer to https://en.wikipedia.org/wiki/List_of_tz_database_time_zones
ntp	String	<i>optional</i>	(Debian/Ubuntu only) The NTP server address.
users	Array	<i>optional</i>	If specified, this contains an array of objects, each object contains the user account information that will be created after OS installation. 0, 1, or multiple users could be specified. If <i>users</i> is omitted, null or empty, no user will be created. See users for more details.
dnsServers	Array	<i>optional</i>	If specified, this contains an array of string, each element is the Domain Name Server, the first one will be primary, others are alternative.
ntpServers	Array	<i>optional</i>	If specified, this contains an array of string, each element is the Network Time Protocol Server.
networkDevices	Array	<i>optional</i>	The static IP setting for network devices after OS installation. If it is omitted, null or empty, RackHD will not touch any network devices setting, so all network devices remain at the default state (usually default is DHCP). If there is multiple setting for same device, RackHD will choose the last one as the final setting, both ipv4 and ipv6 are supported here. (ESXi only) , RackHD will choose the first one in networkDevices as the boot network interface.) See networkDevices for more details.
rootSshKey	String	<i>optional</i>	The public SSH key that will be appended to target OS.
installDisk	String/Number	<i>optional</i>	<i>installDisk</i> is to specify the target disk which the OS will be installed on. It can be a string or a number. For string , it is a disk path that the OS can recognize, its format varies with OS. For example, “/dev/sda” or “/dev/disk/by-id/scsi-36001636121940cc01df404d80c1e761e” for CentOS/RHEL, “t10.ATA_____SATADOM2DSV_3SE_____20130522AA0990120088” or “naa.6001636101840bb01df404d80c2d76fe” or “mpx.vmhba1:C0:T1:L0” or “vml.0000000000766d686261313a313a30” for ESXi. For number , it is a RackHD generated disk identifier (it could be obtained from “driveId” catalog). If <i>installDisk</i> is omitted, RackHD will assign the default disk by order: SATADOM -> first disk in “driveId” catalog -> “sda” for Linux OS. NOTE: Users need to make sure the <i>installDisk</i> (either specified by user or by default) is the first bootable drive from BIOS and raid controller setup. PhotonOS only supports “/dev/sd*” format currently.
installPartitions	Array	<i>optional</i>	<i>installPartitions</i> is to specify the <i>installDisk</i> ’s partitions when OS installer’s default auto partitioning is not wanted. (Only support CentOS at present, Other Linux OS will be supported). See installPartitions for more details.

For **users** in payload:

Parameters	Type	Flags	Description
name	String	required	The name of user. it should start with a letter or digit or underline and the length of it should larger than 1(>=1).
password	String	required	The password of user, it could be clear text, RackHD will do encryption before store it into OS installer's config file. The length of password should larger than 4(>=5). Some OS distributions' password requirements <i>must</i> be satisfied. For ESXi 5.5 , ESXi 5 Password Requirements . For ESXi 6.0 , ESXi 6 Password Requirements .
uid	Number	<i>optional</i>	The unique identifier of user. It should be between 500 and 65535 . (Not support for ESXi OS)
sshKey	String	<i>optional</i>	The public SSH key that will be appended into target OS.

For **networkDevices** in payload, both ipv4 and ipv6 are supported

Parameters	Type	Flags	Description
device	String	required	Network device name (ESXi only , or MAC address) in target OS (ex. "eth0", "enp0s1" for Linux, "vmnic0" or "2c:60:0c:ad:d5:ba" for ESXi)
ipv4	Object	<i>optional</i>	See ipv4 or ipv6 more details.
ipv6	Object	<i>optional</i>	See ipv4 or ipv6 more details.
esxSwitch-Name	String	<i>optional</i>	(ESXi only) The vswitch to attach the vmk device to. vSwitch0 is used by default if no esxSwitchName is specified.

For **installPartitions** in payload:

Parameters	Type	Flags	Description
mount-Point	String	required	Mount point, it could be "/boot", "/", "swap", etc. just like the mount point input when manually installing OS.
size	String	required	Partition size, it could be a number string or "auto", For number, default unit is MB . For "auto", all available free disk space will be used.
fsType	String	<i>optional</i>	File system supported by OS, it could be "ext3", "xfs", "swap", etc. If <i>mountPoint</i> is "swap", the fsType must be "swap".

- **Debian/Ubuntu** installation requires boot, root and swap partitions, make sure your auto sized partition must be the last partition.

For **ipv4** or **ipv6** configurations:

Parameters	Type	Flags	Description
ipAddr	String	required	The assigned static IP address
gateway	String	required	The gateway.
netmask	String	required	The subnet mask.
vlanIds	Array	<i>optional</i>	The VLAN ID. This is an array of integers (0-4095). In the case of Windows OS, the vlan is an array of one parameter only
mtu	Number	<i>optional</i>	Size of the largest network layer protocol data unit

For **switchDevices (ESXi only)** in payload:

Parameters	Type	Flags	Description
switch-Name	String	required	The name of the vswitch
uplinks	String	<i>optional</i>	The array of vmnic# devices or MAC address to set as the uplinks.(Ex: uplinks: ["vmnic0", "2c:60:0c:ad:d5:ba"]). If an uplink is attached to a vSwitch, it will be removed from the old vSwitch before being added to the vSwitch named by 'switchName'.
failover-Policy	String	<i>optional</i>	This can be one of the following options: explicit: Always use the highest order uplink from the list of active adapters which pass failover criteria. iphash: Route based on hashing the src and destination IP addresses mac: Route based on the MAC address of the packet source. portid: Route based on the originating virtual port ID.

For **bonds (RHEL/CentOS only)** in payload:

Parameters	Type	Flags	Description
name	String	required	The name of the bond. Example 'bond0'
nics	Array	<i>optional</i>	The array of server NICs that needs to be included in the bond.
bondvlaninterfaces	Array	<i>optional</i>	List of tagged sub-interfaces to be created associated with the bond interface

For **bondvlaninterfaces** in payload, both ipv4 and ipv6 are supported

Parameters	Type	Flags	Description
vlanid	Number	required	VLAN ID to be associated with the tagged sub interface
ipv4	Object	<i>optional</i>	See ipv4 or ipv6 more details.
ipv6	Object	<i>optional</i>	See ipv4 or ipv6 more details.

Windows OS Installation Workflow Payload

Parameters	Type	Flags	Description
productkey	String	required	Windows License
domain	String	optional	Windows domain
hostname	String	optional	Windows hostname to be giving to the node after installation
smbUser	String	required	Smb user for the share to which Windows' iso is mounted
smbPassword	String	required	Smb password
repo	String	required	The share to to which Windows' iso is mounted

Example of minimum payload https://github.com/RackHD/RackHD/blob/master/example/samples/install_windows_payload_minimal.json

Example of full payload https://github.com/RackHD/RackHD/blob/master/example/samples/install_windows_payload_full.json

RAID Configuration

RackHD supports RAID configuration to create and delete RAID for hardwares with LSI RAID controller.

Create docker image with Storcli/Perccli

RackHD leverages LSI provided tool Storcli to configure RAID. RackHD requires user to build docker image including Storcli. As on how to build docker image for RackHD, please refer to <https://github.com/RackHD/on-imagebuilder>. Perccli is a Dell tool which is based on Storcli and has the same commands with it. If user wants to configure RAID on Dell servers, Perccli instead of Storcli should be built in docker image. The newly built docker image(default named “dell.raid.docker.tar.xz” for Dell and “raid.docker.tar.xz” for others) should be put in RackHD static file path.

Create RAID

An example of creating RAID workflow is as below:

```
curl -X POST \
  -H 'Content-Type: application/json' \
  -d @params.json \
  <server>/api/current/nodes/<identifier>/workflows?name=Graph.Raid.Create.MegaRAID'
```

An example of params.json with minimal parameters for creating RAID workflow:

```
{
  "options": {
    "bootstrap-rancher": {
      "dockerFile": "raid.docker.tar.xz"
    },
    "create-raid": {
      "raidList": [
        {
          "enclosure": 255,
          "type": "raid1",
          "drives": [1, 4],
          "name": "VD0"
        },
        {
          "enclosure": 255,
          "type": "raid5",
          "drives": [2, 5, 3],
          "name": "VD1"
        }
      ]
    }
  }
}
```

For details on items of create-raid.options, please refer to: <https://github.com/RackHD/on-tasks/blob/master/lib/task-data/schemas/create-megaraid.json>.

Note:

- User need make sure drives are under UGOOD status before creating RAID. If drives are under other status (JBOD, online/offline or UBAD), RackHD won't be able to create RAID with them.
- For Dell servers, tool path in docker container should be specified in param.json as below:

```
{
  "options": {
    "bootstrap-rancher": {
      "dockerFile": "dell.raid.docker.tar.xz"
    },
    "create-raid": {
      "path": "/opt/MegaRAID/perccli/percli64",
      "raidList": [
        {
          "enclosure": 255,
          "type": "raid1",
          "drives": [1, 4],
          "name": "VD0"
        },
        {
          "enclosure": 255,
          "type": "raid5",
          "drives": [2, 5, 3],
          "name": "VD1"
        }
      ]
    }
  }
}
```

Delete RAID

An example of deleting RAID workflow is as below:

```
curl -X POST \
-H 'Content-Type: application/json' \
-d @params.json \
<server>/api/current/nodes/<identifier>/workflows?name=Graph.Raid.Delete.MegaRAID'
```

An example of params.json for deleting RAID workflow:

```
{
  "options": {
    "delete-raid": {
      "raidIds": [0, 1]
    },
    "bootstrap-rancher": {
      "dockerFile": "raid.docker.tar.xz"
    }
  }
}
```

“raidIds” is the virtual disk id to be deleted.

For Dell servers, the payload should look like:

```
{
  "options": {
    "delete-raid": {
```

```

        "path": "/opt/MegaRAID/perccli/percli64",
        "raidIds": [0, 1]
    },
    "bootstrap-rancher": {
        "dockerFile": "dell.raid.docker.tar.xz"
    }
}
}

```

Disk Secure Erase

Secure Erase (SE) also known as a wipe is to destroy data on a disk so that data can't or is difficult to be retrieved. RackHD implements solution to do disk Secure Erase.

Disk Secure Erase Workflow API

An example of starting secure erase for disks:

```

curl -X POST \
  -H 'Content-Type: application/json' \
  -d @params.json \
  <server>/api/current/nodes/<identifier>/workflows?name=Graph.Drive.SecureErase

```

An example of params.json for disk secure erase:

```

{
  "options": {
    "drive-secure-erase": {
      "eraseSettings": [
        {
          "disks": ["sdb"],
          "tool": "sg_format",
          "arg": "0"
        },
        {
          "disks": ["sda"],
          "tool": "scrub",
          "arg": "nnsa"
        }
      ]
    },
    "disk-scan-delay": {
      "duration": 10000
    }
  }
}

```

Use below command to check the workflow is active or inactive:

```

curl <server>/api/current/nodes/<identifier>/workflows?active=true

```

Deprecated 1.1 API - Use below command to check the workflow is active or inactive:

```

curl <server>/api/1.1/nodes/<identifier>/workflows/active

```

Use below command to stop the active workflow to cancel secure erase workflow:

```
curl -X PUT \
-H 'Content-Type: application/json' \
-d '{"command": "cancel"}' \
<server>/api/current/nodes/<id>/workflows/action
```

Deprecated 1.1 API - Use below command to stop the active workflow to cancel secure erase workflow:

```
curl -X DELETE <server>/api/1.1/nodes/<identifier>/workflows/active
```

Disk Secure Erase Workflow Payload

Parameters descriptions of secure erase workflow payload are listed below. Among them, *duration* is for *drive-scan-delay* task, other parameters are for *drive-secure-erase* task.

Parameters	Type	Flags	Description
eraseSettings	Array	required	Contains secure erase option list, each list element is made up of “disks” and optional “tool” and “arg” parameters.
disks	Array	required	Contains disks to be erased, both devName or identifier from driveId catalog are eligible.
tool	String	optional	Specify tool to be used for secure erase. Default it would be scrub.
arg	String	optional	Specify secure erase arguments with specified tools.
duration	Integer	optional	Specify delay time in milliseconds. After node boots into microkernel, it takes some time for OS to scan all disks. <i>duration</i> is designed so that secure erase is initiated after all disks are scanned. <i>duration</i> is 10 seconds if not specified.

Supported Disk Secure Erase Tools

RackHD currently supports disk secure erase with four tools: scrub, hdparm, sg_sanitize, sg_format. If “tool” is not specified in payload, “scrub” is used as default. Below table includes description for different tools.

Tool	Description
scrub	Scrub iteratively writes patterns on files or disk devices to make retrieving the data more difficult. Scrub supports almost all drives including SATA, SAS, USB and so on.
hdparm	Hdparm can be used to issue ATA instruction of Secure Erase or enhanced secure erase to a disk. Hdparm works well with SATA drives, but it can brick a USB drive if it doesn’t support SAT (SCSI-ATA Command Translation).
sg_sanitize	Sg_sanitize (from sg3-utils package) removes all user data from disk with SCSI SANITIZE command. Sanitize is more likely to be implemented on modern disks (including SSDs) than FORMAT UNIT’s security initialization feature and in some cases much faster. However since it is relative new and optional, not all SCSI drives support SANITIZE command
sg_format	Sg_format (from sg3-utils package) formats, resizes or modifies protection information of a SCSI disk. The primary goal of a format is the configuration of the disk at the end of a format (e.g. different logical block size or protection information added). Removal of user data is only a side effect of a format.

Supported Disk Secure Erase Arguments

Default argument for scrub is “nnsa”, below table shows supported arguments for **scrub** tool:

Supported args	Description
nnsa	4-pass NNSA Policy Letter NAP-14.1-C (XVI-8) for sanitizing removable and non-removable hard disks, which requires overwriting all locations with a pseudorandom pattern twice and then with a known pattern: random(x2), 0x00, verify. scrub default arg=nnsa
dod	4-pass DoD 5220.22-M section 8-306 procedure (d) for sanitizing removable and non-removable rigid disks which requires overwriting all addressable locations with a character, its complement, a random character, then verify. NOTE: scrub performs the random pass first to make verification easier: random, 0x00, 0xff, verify.
bsi	9-pass method recommended by the German Center of Security in Information Technologies (http://www.bsi.bund.de): 0xff, 0xfe, 0xfd, 0xfb, 0xf7, 0xef, 0xdf, 0xbf, 0x7f.
fillzero	1-pass pattern: 0x00.
fillff	1-pass pattern: 0xff.
random	1-pass pattern: random(x1).
random2	2-pass pattern: random(x2).
custom=0xdd	1-pass custom pattern.
gutmann	The canonical 35-pass sequence described in Gutmann's paper cited below.
schneier	7-pass method described by Bruce Schneier in "Applied Cryptography" (1996): 0x00, 0xff, random(x5)
pfitzner7	Roy Pfitzner's 7-random-pass method: random(x7).
pfitzner33	Roy Pfitzner's 33-random-pass method: random(x33).
old	6-pass pre-version 1.7 scrub method: 0x00, 0xff, 0xaa, 0x00, 0x55, verify.
fastold	5-pass pattern: 0x00, 0xff, 0xaa, 0x55, verify.
usarmy	US Army AR380-19 method: 0x00, 0xff, random. The same with dod option

Default argument for `hdparm` is "security-erase", below table shows supported arguments for **hdparm** tool:

Supported args	Description
security-erase	Issue ATA Secure Erase (SE) command. <code>hdparm</code> default arg="security-erase"
security-erase-enhanced	Enhanced SE is more aggressive in that it ought to wipe every sector: normal, HPA, DCO, and G-list. Not all drives support this command

Default argument for `sg_sanitize` is "block", below table shows supported arguments for **sg_sanitize** tool:

Supported args	Description
block	Perform a "block erase" sanitize operation. <code>sg_sanitize</code> default arg="block"
fail	Perform an "exit failure mode" sanitize operation.
crypto	Perform a "cryptographic erase" sanitize operation.

Default argument for `sg_format` is "1", below table shows supported arguments for **sg_format** tool:

Supported args	Description
"1"	Disable Glist erasing. <code>sg_format</code> default arg="1"
"0"	Enable Glist erasing

Disk Secure Erase Workflow Notes

Please pay attention to below items if you are using RackHD secure erase function:

- **RackHD Secure Erase is not fully tested.** RackHD secure erase is tested on RackHD supported servers with only one LSI RAID controller. Servers with multiple RAID controllers, disk array enclosures or non-LSI RAID controllers are not tested.

- **Use RackHD to manage RAID operation.** RackHD relies on its catalog data for secure erase. If RAID operation is not done via RackHD, RackHD secure erase workflow might not be able to recognize drive names given and fail. A suggestion is to re-run discovery for the compute node if you did changed RAID configure not using RackHD.
- **Secure Erase is time-consuming.** Hdparm, sg_format and sg_sanitize will leverage drive firmware to do secure erase, even so it might take hours for a 1T drive. Scrub is overwriting data to disks and its speed is depends on argument you chose. For a “gutmann” argument, it will take days to erase a 1T drive.
- **Cancel Secure Erase workflow can’t cancel secure erase operation.** Hdparm, sg_sanitize and sg_format are leverage drive firmware to do secure erase, once started there is no proper way to ask drive firmware to stop it till now.
- **Power cycle is risky.** Except for scrub tool, other tools are actually issue a command to drive and drive itself will control secure erase. That means once you started secure erase workflow, you can’t stop it until it is completed. If you power cycled compute node under this case, drive might be frozen, locked or in worst case bricked. All data will not be accessible. If this happens, you need extra effort to bring your disks back to normal status.

Firmware Update

Firmware update Example using SKU Pack

This example provides instructions on how to flash a BMC image on a Quanta (node) using SKU Pack.

1. Wait for discovery to complete and get nodes to check if node has been discovered successfully

Get Nodes

```
GET /api/current/nodes
```

```
curl <server>/api/current/nodes
```

2. Post the obm settings if they don’t already exist for the node. An example on how to do this is shown in Section 7.1.8.1 here <http://rackhd.readthedocs.io/en/latest/tutorials/vagrant.html#adding-a-sku-definition> Section 7.1.8.1
3. Acquire BMC files and utilities from the vendor. Go to the Quanta directory, a sub-directory of the root folder of on-skupack, extract the BMC image and BMC upgrade executable into the static/bmc of the skupack and update the config.json with the md5sum of the firmware image.
4. The firmware files and update utilities need to be built into a SKU package

Build SKU Package

```
$ ./build-package.bash <sku_pack_directory> <subname>
```

<sku_pack_directory> must be one of the directory names containing the node type on the root directory of on-skupack, e.g., it can be quanta-d51-1u, quanta-t41,dell-r630, etc, and <subname> can be any name a user likes. A {sku_pack_directory_subname}.tar.gz will be created in tarballs folder of the same directory.

```
$ ls ./tarballs
sku_pack_directory_subname.tar.gz
```

5. The SKU package that was built needs to be registered

POST the tarball


```
curl -X POST --data-binary @tarballs/sku_pack_directory_subname.tar.gz localhost:8080/api/cu
```

The above command will return a SKU ID. If an error like “Duplicate name found” is returned in place of the SKU ID, check the database and delete the preexisting SKU package.

- The pollers associated with the node need to be paused before POST’ing the Workflow to flash a new BMC image. This is needed to avoid seeing any poller errors in the log while BMC is offline. Further information on IPMI poller properties can be found at [Pollers](#)

Get List of Active Pollers Associated With a Node

```
GET /api/current/nodes/:id/pollers
```

```
curl <server>/api/current/nodes/<nodeid>/pollers
```

Update a Single Poller to pause the poller

```
PATCH /api/current/pollers/:id
{
  "paused": true
}
```

```
curl -X PATCH \
-H 'Content-Type: application/json' \
-d '{"paused":true}' \
<server>/api/current/pollers/<pollerid>
```

- The workflow to flash a new BMC image to a Quanta node needs to be POST’ed If a user would upgrade a node without reboot at the end or run BMC upgrade with a file override, a user need add a payload when posting the workflow. Details please refer to the README.md under Quanta directory.

POST Workflow

```
POST /api/current/nodes/:id/workflows?name=Graph.Flash.Quanta.Bmc
```

```
curl -X POST <server>/api/current/nodes/<nodeid>/workflows?name=Graph.Flash.Quanta.Bmc
```

- Check if any active workflows on that node exist to make sure the workflow has completed

GET active Workflow

```
GET /api/current/nodes/<id>/workflows/active
```

```
curl <server>/api/current/nodes/<id>/workflows/active
```

If a remote viewing session exists for the node, check the BMC firmware to verify the version has been updated.

Southbound Notification API

The southbound notification API provides functionality for sending notifications to RackHD from a node. For example, a node could send notification to inform RackHD that OS installation has finished.

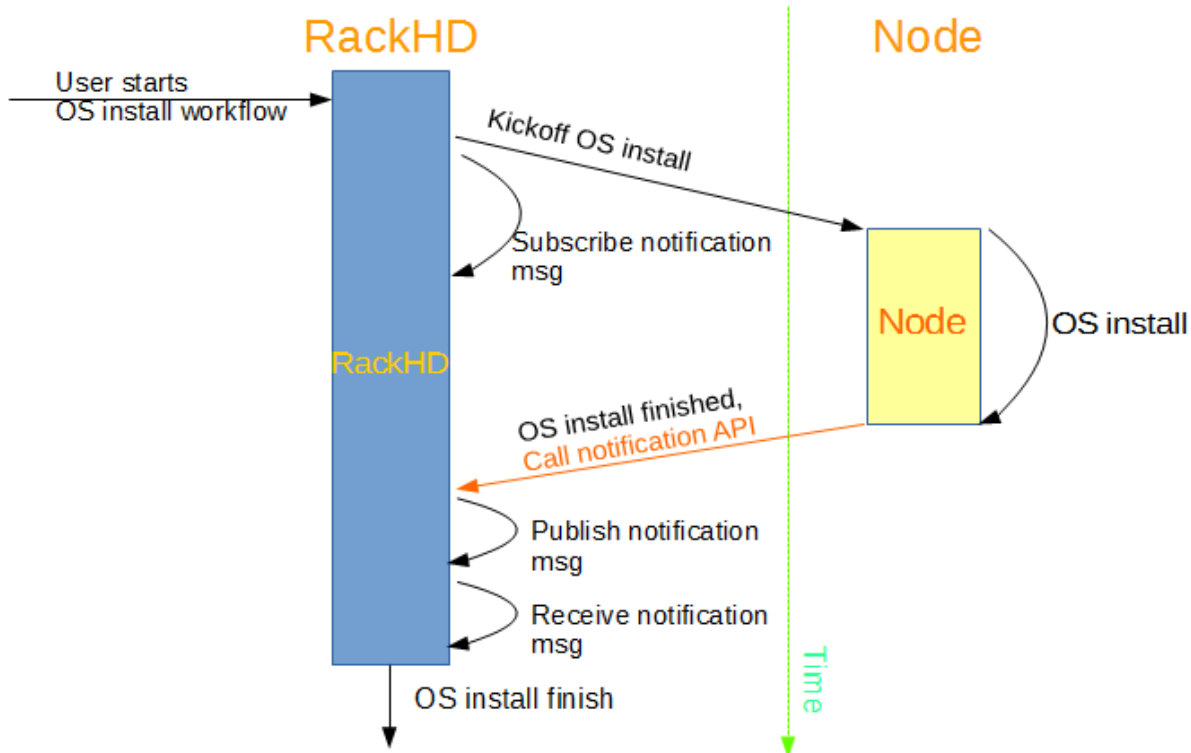
The notification API is only available from the southbound.

How does it work

When a node calls a notification API, the RackHD on-http process will get acknowledged and then send a AMQP message to an exchange named 'on.events', with routing key set to 'notification' or 'notification.<id>' depending on the parameters sent along when calling the notification API.

Any task running in on-taskgraph process that is expecting a notification will need to subscribe the AMQP message.

For example, the install-os task will subscribe the 'on.events' AMQP message with routing key 'notification.<id>'. A node will call the notification API at the end of the OS installation thus on-http will publish a AMQP message accordingly. The install-os task will then receive the message and finish itself. Please refer to the diagram below.



API commands

When running the on-http process, these are some common API commands you can send:

Send notification targeting a node

```
POST /api/current/notification?nodeId=<id>
```

```
curl -X POST -H "Content-Type:application/json" \
<server>/api/current/notification?nodeId=5542b78c130198aa216da3ac
```

It will also work if the nodeId parameter is set in the request body.

```
curl -X POST -H "Content-Type:application/json" <server>/api/current/notification \
-d '{"nodeId": "5542b78c130198aa216da3ac"}'
```

Additional parameters can be sent as well, as long as the receiver task knows how to use those parameters.

```
curl -X POST -H "Content-Type:application/json" \
<server>/api/current/notification?nodeId=5542b78c130198aa216da3ac \
&progress=50%status=inprogress
```

Send a broadcast notification

A broadcast notification will trigger a AMQP message with routing key set to 'notification', without the trailing '<id>'.

```
POST /api/current/notification
```

```
curl -X POST -H "Content-Type:application/json" <server>/api/current/notification
```

Use notification API in OS installation

A typical OS installation needs two notifications. The first one notifies that OS has been installed to the disk on the target node. The second one notifies that the OS has been successfully booted on the target node.

The first notification is typically sent in the 'postinstall' section of the kickstart file. For example: <https://github.com/RackHD/on-http/blob/master/data/templates/install-photon/photon-os-ks#L76>

the second notification is typically sent in the RackHD callback script. For example: <https://github.com/RackHD/on-http/blob/master/data/templates/install-photon/photon-os.rackhdcallback#L38>

MicroKernel image

RackHD utilizes [RancherOS](#) booted in RAM and a customized docker image run in RancherOS to perform various operations such as node discovery and firmware management.

The [on-imagebuilder](#) repository contains a set of scripts that uses [Docker](#) to build docker images that run in RancherOS, primarily for use with the [on-taskgraph](#) workflow engine.

Requirements

- Docker

Bootstrap Process

The images produced by these scripts are intended to be netbooted and run in RAM. The typical flow for how these images are used/booted is this:

- Netboot **RancherOS** (kernel and initrd) via PXE/iPXE
- The custom cloud-config file requests a **rackhd/micro** docker image from the boot server.
- It then starts a container with full container capabilities using the **rackhd/micro** docker image.

Building Images

Instructions for building images, can be found in the [on-imagebuilder README](#).

How To Login Microkernel

By default, RackHD has a workflow to let users login RancherOS based microkernel to debug. The workflow name is *Graph.BootstrapRancher*.

```
curl -X POST -H 'Content-Type: application/json' <server>/api/current/nodes/<identifier>/workflows?name=Graph.BootstrapRancher
```

When this workflow is running, it will set node to PXE boot, then reboot the node. The node will boot into microkernel, finally you could SSH login node's microkernel from the RackHD server. The node's IP address could be retrieved from 'GET /lookups' API like below, the SSH username:password is *rancher:monorail*.

```
curl <server>/api/current/lookups?q=<identifier>
```

Service Setup

TFTP and DHCP Service Setup

RackHD is flexible to adapt to different network environments for TFTP and DHCP service. By default, RackHD use *on-tftp* for TFTP service, *ISC DHCP Server* and DHCP proxy *on-dhcp-proxy* for DHCP service, and they are deployed in RackHD server along with other RackHD service *on-http*, *on-taskgraph*, *on-syslog*. They could be replaced with other TFTP and DHCP services, and also could be deployed to a separate server.

Cases	Supported TFTP Service	Supported DHCP Service
TFTP and DHCP services are provided from the RackHD server	<ol style="list-style-type: none"> 1. on-tftp(default) 2. Third-party TFTP service such as in.tftpd(tftp-hpa) in Ubuntu OS 	<ol style="list-style-type: none"> 1. ISC DHCP + on-dhcp-proxy(default) 2. ISC DHCP only 3. Third-party DHCP Service + DHCP proxy 4. Third-party DHCP Service only
TFTP and DHCP services are provided from a separate server	<ol style="list-style-type: none"> 1. on-tftp 2. Third-party TFTP service such as in.tftpd(tftp-hpa) in Ubuntu OS 	<ol style="list-style-type: none"> 1. ISC DHCP + on-dhcp-proxy 2. ISC DHCP only 3. Third-party DHCP Service + DHCP proxy 4. Third-party DHCP Service only

NOTE: "Third-party" service means it's not the RackHD default service.

TFTP and DHCP from the RackHD Server

TFTP Service Configuration in the RackHD Server Default on-tftp Configuration

The RackHD default TFTP service is *on-tftp*, it could be configured by fields *tftpBindAddress*, *tftpBindPort*, *tftpRoot* in *config.json*, and RackHD iPXE files are placed into the *tftpRoot* directory.

```
...
"tftpBindAddress": "172.31.128.1",
"tftpBindPort": 69,
"tftpRoot": "./static/tftp",
...
```

Third-Party TFTP Service Configuration

In many cases, another TFTP service can be used with RackHD. RackHD simply needs the files that on-tftp would serve to be provided by another instance of TFTP. You can frequently do this by simply placing the [RackHD iPXE](#) files into the TFTP service root directory.

For scripts in [RackHD TFTP Templates](#), where the parameters such as *apiServerAddress*, *apiServerPort* are rendered by *on-tftp*, they need to be hardcoded, They are *172.31.128.1* and *9080* in the example, then provide these scripts into the TFTP root directory.

NOTE:

1. If all managed nodes' NIC ROM are iPXE, not PXE, then you don't need to provide [RackHD iPXE](#) files into the TFTP directory.
2. If the functionality supported by rendered scripts is not needed, then you don't need to provide [RackHD TFTP Templates](#) scripts into the TFTP directory.
3. If both cases above are satisfied, the TFTP service is not needed by RackHD.

DHCP Service Configuration in the RackHD Server The DHCP protocol is a critical component to the PXE boot process and for executing various profiles and [Workflow Graphs](#) within RackHD.

By default RackHD deploys a DHCP configuration that forwards DHCP clients to the on-dhcp-proxy service, see [System Architecture](#) for more information. However conventional DHCP configurations that require static (and/or dynamic) IP lease reservations are also supported, bypassing the on-dhcp-proxy service all together.

There are various DHCP Server versions out there, RackHD has been primarily validated against [ISC DHCP Server](#). As long as the DHCP server supports the required DHCP configuration options then those versions should be compatible.

Default ISC DHCP + on-dhcp-proxy Configuration

The advantage of using the on-dhcp-proxy service is to avoid complication DHCP server setup, most of the logic is handled in on-dhcp-proxy, it's convenient and flexible. A typical simple *dhcpd.conf* of [ISC DHCP Server](#) for forwarding DHCP request to RackHD's on-dhcp-proxy service would work like the following:

```
ddns-update-style none;
option domain-name "example.org";
option domain-name-servers ns1.example.org, ns2.example.org;

default-lease-time 600;
max-lease-time 7200;
log-facility local7;

deny duplicates;

ignore-client-uids true;

subnet 172.31.128.0 netmask 255.255.240.0 {
    range 172.31.128.2 172.31.143.254;
    # Use this option to signal to the PXE client that we are doing proxy DHCP
    # Even not doing proxy DHCP, it's essential, otherwise, monorail-undionly.kpxe
    # would not DHCP successfully.
    option vendor-class-identifier "PXEClient";
}
```

Substituting the [subnet](#), [range](#) and [netmask](#) to match your desired networking configuration.

To enforce lease assignment based on MAC and not UID we opt-in to ignore the UID in the request by setting **ignore-client-uids true**.

ISC DHCP Only Configuration

ISC DHCP service can also define static host definitions, and not use on-dhcp-proxy. It would work like the following:

```
ddns-update-style none;

option domain-name "example.org";
option domain-name-servers ns1.example.org, ns2.example.org;

default-lease-time 600;
max-lease-time 7200;

log-facility local7;

deny duplicates;
ignore-client-uids true;

option arch-type code 93 = unsigned integer 16;

subnet 172.31.128.0 netmask 255.255.240.0 {
    range 172.31.128.2 172.31.143.254;
    next-server 172.31.128.1;

    # It's essential for Ubuntu installation
    option routers 172.31.128.1;
    # It's essential for Ubuntu installation
    option domain-name-servers 172.31.128.1;

    # It's essential, otherwise, monorail-undionly.kpxe would not DHCP successfully.
    option vendor-class-identifier "PXECClient";

    # Register leased hosts with RackHD
    if ((exists user-class) and (option user-class = "MonoRail")) {
        filename "http://172.31.128.1:9080/api/current/profiles";
    } else {
        if option arch-type = 00:09 {
            filename "monorail-efi64-snpnonly.efi";
        } elseif option arch-type = 00:07 {
            filename "monorail-efi64-snpnonly.efi";
        } elseif option arch-type = 00:06 {
            filename "monorail-efi32-snpnonly.efi";
        } elseif substring(binary-to-ascii(16, 8, ":", substring(hardware, 1, 6)), 0, 8) = "0:2:c9" {
            # If the mac belongs to a mellanox card, assume that it already has
            # Flexboot and don't hand down an iPXE rom
            filename "http://172.31.128.1:9080/api/current/profiles";
        } elseif substring(binary-to-ascii(16, 8, ":", substring(hardware, 1, 6)), 0, 8) = "ec:a8:6b" {
            filename "monorail.intel.ipxe";
        } elseif substring(option vendor-class-identifier, 0, 6) = "Arista" {
            # Arista skips the TFTP download step, so just hit the
            # profiles API directly to get a profile from an active task
            # if there is one
            filename = concat("http://172.31.128.1:9080/api/current/profiles?macs=", binary-to-ascii(16, 8, ":", substring(hardware, 1, 6)), "0:2:c9");
        } elseif substring(option vendor-class-identifier, 0, 25) = "PXECClient:Arch:00000:UNDI" {
            filename "monorail-undionly.kpxe";
        } else {
            filename "monorail.ipxe";
        }
    }
}
```

```
# Example register static entry lookup with RackHD
host My_Host_SNXYZ {
    hardware ethernet 00:0A:0B:0C:0D:0E;
    fixed-address 172.31.128.120;
    option routers 172.31.128.1;
    if ((exists user-class) and (option user-class = "MonoRail")) {
        filename "http://172.31.128.1:9080/api/common/profiles";
    } else {
        filename "monorail.ipxe";
    }
}
```

In the global subnet definition we define a PXE chainloading setup to handle specific client requests.

```
if ((exists user-class) and (option user-class = "MonoRail")) {
    ...
} else {
    ...
}
```

If the request is made from a BIOS/UEFI PXE client, the DHCP server will hand out the iPXE bootloader image that corresponds to the system's architecture type.

```
if ((exists user-class) and (option user-class = "MonoRail")) {
    filename "http://172.31.128.1:9080/api/current/profiles";
} else {
    if option arch-type = 00:09 {
        filename "monorail-efi64-snponly.efi";
    } elsif option arch-type = 00:07 {
        filename "monorail-efi64-snponly.efi";
    } elsif option arch-type = 00:06 {
        filename "monorail-efi32-snponly.efi";
    } else {
        filename "monorail.ipxe";
    }
}
```

If the request is made from the RackHD iPXE client, the DHCP server will chainload another boot configuration pointed at RackHD's profiles API.

Third-Party DHCP Service Configuration

The third-party DHCP service could be used with possible solution configurations below:

Service	Cases	Solutions
Third-party DHCP service only	DHCP service has functionalities like ISC DHCP, it could configure DHCP to return different bootfile name according to <i>user-class</i> , <i>arch-type</i> , <i>vendor-class-identifier</i> etc.	Configure it like ISC DHCP to make node auto chainloading iPXE files and finally iPXE hit RackHD URL <code>http://172.31.128.1:9080/api/current/profiles</code> IP address and port are configured according to RackHD southbound configuration.
	DHCP service could not proxy DHCP, and on-dhcp-proxy also could not be deployed in the DHCP server, only bootfile name could be specified by DHCP	Replace “autoboot” command in Default iPXE Config with “dhcp” and “ <code>http://172.31.128.1:9080/api/current/profiles</code> ”, then re-compile iPXE in on-imagebuilder to generate new iPXE files, specify one of generated iPXE files as bootfile name in DHCP configuration. IP address and port are configured according to RackHD southbound configuration. Two drawbacks for this solution due to DHCP and environment limitations: 1. IP address and port are hardcoded in iPXE file 2. Only one iPXE bootfile name could be specified. it's not flexible to switch bootfile name automatically.
Third-party DHCP service + DHCP proxy	DHCP service's functionality is less than ISC DHCP, but it could proxy DHCP like ISC DHCP's configuration “option vendor-class-identifier “PXEClient”	on-dhcp-proxy could be leveraged to avoid complicated DHCP configuration.

TFTP and DHCP from a Separate Server

The RackHD default TFTP and DHCP services such as on-tftp, on-dhcp-proxy and ISC DHCP could be deployed in a separate server with some simple configurations.

RackHD also could work without its own TFTP and DHCP service, and leverage an existing TFTP and DHCP server from the datacenter or lab environments.

When TFTP and DHCP are installed in a separate server, both the RackHD server and the TFTP/DHCP server need to be set.

NOTE: TFTP and DHCP server IP address is **172.31.128.1**, and RackHD server IP address is **172.31.128.2** in the example below.

RackHD Main Services Configuration in the RackHD Server In the RackHD server, `/opt/monorail/config.json` is updated with settings below, then restart `on-http`, `on-taskgraph` and `on-syslog` services.

```
...
"apiServerAddress": "172.31.128.2",
...
"syslogBindAddress": "172.31.128.2"
...
"dhcpGateway": "172.31.128.1",
"dhcpProxyBindAddress": "172.31.128.1",
...
"tftpBindAddress": "172.31.128.1",
...
```



```
"httpEndpoints": [
  ...
  {
    ...
    "address": "172.31.128.2",
    ...
  },
  ...
]
...
```

TFTP Service Configuration in the Separate Server Default on-tftp Configuration

/opt/monorail/config.json need to be updated with settings below, then restart *on-tftp*.

```
...
"apiServerAddress": "172.31.128.2",
...
"syslogBindAddress": "172.31.128.2"
...
"dhcpGateway": "172.31.128.1",
"dhcpProxyBindAddress": "172.31.128.1",
...
"tftpBindAddress": "172.31.128.1",
...
"httpEndpoints": [
  ...
  {
    ...
    "address": "172.31.128.2",
    ...
  },
  ...
]
...
```

Third-Party TFTP Service Configuration

The third-party TFTP service setup in the separate server is the same with in RackHD server. [RackHD TFTP Templates](#) scripts' rendered parameters *apiServerAddress*, *apiServerPort* is *172.31.128.2*, *9080* in the example.

DHCP Service Configuration in the Separate Server Default ISC DHCP + on-dhcp-proxy Configuration

ISC DHCP *dhcpd.conf* need to be updated with settings below, then restart ISC DHCP. **NOTE:** DHCP ip addresses range starts from **172.31.128.3**, because **172.31.128.2** is assigned to RackHD server.

```
ddns-update-style none;
option domain-name "example.org";
option domain-name-servers ns1.example.org, ns2.example.org;

default-lease-time 600;
max-lease-time 7200;
log-facility local7;

deny duplicates;

ignore-client-uids true;
```

```
subnet 172.31.128.0 netmask 255.255.240.0 {
  range 172.31.128.3 172.31.143.254;
  # Use this option to signal to the PXE client that we are doing proxy DHCP
  # Even not doing proxy DHCP, it's essential, otherwise, monorail-undionly.kpxe
  # would not DHCP successfully.
  option vendor-class-identifier "PXEClient";
}
```

`/opt/monorail/config.json` need to be updated with settings below, then restart *on-dhcp-proxy*.

```
...
"apiServerAddress": "172.31.128.2",
...
"syslogBindAddress": "172.31.128.2"
...
"dhcpGateway": "172.31.128.1",
"dhcpProxyBindAddress": "172.31.128.1",
...
"tftpBindAddress": "172.31.128.1",
...
"httpEndpoints": [
  ...
  {
    ...
    "address": "172.31.128.2",
    ...
  },
  ...
]
...
```

ISC DHCP Only Configuration

ISC DHCP `dhcpd.conf` need to be updated with settings below, then restart ISC DHCP. **NOTE:** DHCP ip addresses range starts from **172.31.128.3**, because **172.31.128.2** is assigned to RackHD server.

```
ddns-update-style none;

option domain-name "example.org";
option domain-name-servers ns1.example.org, ns2.example.org;

default-lease-time 600;
max-lease-time 7200;

log-facility local7;

deny duplicates;
ignore-client-uids true;

option arch-type code 93 = unsigned integer 16;

subnet 172.31.128.0 netmask 255.255.240.0 {
  range 172.31.128.3 172.31.143.254;
  next-server 172.31.128.1;

  # It's essential for Ubuntu installation
  option routers 172.31.128.1;
  # It's essential for Ubuntu installation
  option domain-name-servers 172.31.128.1;
```

```
# It's essential, otherwise, monorail-undionly.kpxe would not DHCP successfully.
option vendor-class-identifier "PXEClient";

# Register leased hosts with RackHD
if ((exists user-class) and (option user-class = "MonoRail")) {
    filename "http://172.31.128.2:9080/api/current/profiles";
} else {
    if option arch-type = 00:09 {
        filename "monorail-efi64-snponly.efi";
    } elseif option arch-type = 00:07 {
        filename "monorail-efi64-snponly.efi";
    } elseif option arch-type = 00:06 {
        filename "monorail-efi32-snponly.efi";
    } elseif substring(binary-to-ascii(16, 8, ":", substring(hardware, 1, 6)), 0, 8) = "0:2:c9" {
        # If the mac belongs to a mellanox card, assume that it already has
        # Flexboot and don't hand down an iPXE rom
        filename "http://172.31.128.2:9080/api/current/profiles";
    } elseif substring(binary-to-ascii(16, 8, ":", substring(hardware, 1, 6)), 0, 8) = "ec:a8:6b" {
        filename "monorail.intel.ipxe";
    } elseif substring(option vendor-class-identifier, 0, 6) = "Arista" {
        # Arista skips the TFTP download step, so just hit the
        # profiles API directly to get a profile from an active task
        # if there is one
        filename = concat("http://172.31.128.2:9080/api/current/profiles?macs=", binary-to-ascii(16, 8, ":", substring(hardware, 1, 6)), "0:2:c9");
    } elseif substring(option vendor-class-identifier, 0, 25) = "PXEClient:Arch:00000:UNDI" {
        filename "monorail-undionly.kpxe";
    } else {
        filename "monorail.ipxe";
    }
}

# Example register static entry lookup with RackHD
host My_Host_SNXYZ {
    hardware ethernet 00:0A:0B:0C:0D:0E;
    fixed-address 172.31.128.120;
    option routers 172.31.128.1;
    if ((exists user-class) and (option user-class = "MonoRail")) {
        filename "http://172.31.128.2:9080/api/common/profiles";
    } else {
        filename "monorail.ipxe";
    }
}
}
```

Third-Party DHCP Service Configuration

The solutions of using the third-party DHCP service in a separate server are the same with in the RackHD server. Just need to specify RackHD southbound IP address and port in DHCP configuration. they are *172.31.128.2, 9080* in the example.

Static File Service Setup

There are two kinds of static files in RackHD: one of them are used for RackHD functionality, and the other is node discovery and os installation. This section introduces a mechanism to move the latter type to a separate third-party service in order to offload the burden of file transmission in RackHD.

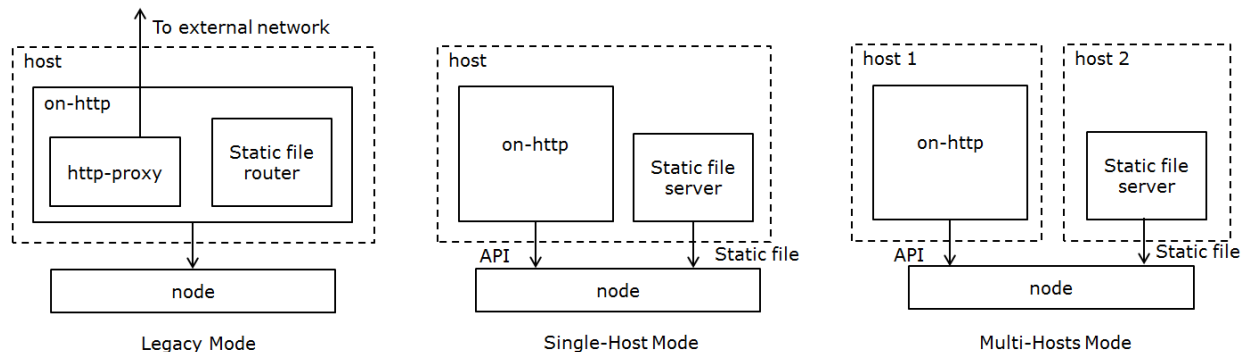
Files That can be Moved into a Separate Server

Some files, including schema, swagger configuration and others, interacts closely with RackHD, and are part of its functionalities. Others are served for node discovery and OS installation (if users put OS image under the same static file directory). `on-http` manages all the files mentioned above by default, and the latter (**files for discovery and OS installation**) can be moved to a third-party static file server, which will be discussed below.

Diagrams for Different Working Modes

RackHD supports three modes to serve static files. This chapter introduces the settings for the last two modes.

- Legacy Mode: nodes get static files from `on-http` service (default).
- Single-Host Mode: nodes get static files from another service in the same host as RackHD.
- Multi-Host Mode: nodes get static files from different host.



Setup a Static File Server

Prerequisites

The server can be accessed by nodes.

Configure a Third-Party Static File Server

Since RackHD doesn't require any customization on a file server, users could adopt any frameworks they are familiar with. Here takes `nginx` as an example about the configuration.

After `install nginx`, modify `nginx_conf` to make sure the following configuration works.

```
http {
    server {
        listen 3000;
        sendfile on;

        location / {
            root /home/onrack/;
        }
    }
}
```

“3000” is the port for the server; “location” is the URI root path to access static files; and “root” specifies the directory that will be used to search for files.

Restart `nginx` server after the new configuration.

Copy Static File into the Server

In the RackHD file directory on static file server (specified in “root” item above), create a directory named “common”. Copy files from [on-imagebuilder binary in bintray](#) into this folder.

Configure the Path of Static File Server in RackHD

In `config.json`, add the following fields:

```
...
"fileServerAddress": "172.31.128.3",
"fileServerPort": 3000,
"fileServerPath": "/",
...
```

The following table describes the configurations above.

Parameter	Description
fileServerAddress	IP address of static file server that nodes can access
fileServerPort	port the server is listening to. <i>Optional</i> , the default value is 80
fileServerPath	the “location” in server configuration. <i>Optional</i> , the default value is ‘/’

Restart RackHD services after adding these fields.

Notes

- fileServer configurations takes higher priority than [httpStaticRoot](#), which means that when above fields exists, RackHD will use file server address for static files and ignore that specified “httpStaticRoot”.
- When user creates a payload for a task, they could use `{{ file.server }}` as the address that nodes will use to get static file. It will direct to the correct address holding static file, depending on different working modes.
- [httpProxies](#) still works. If user has setup a static file server, but would like to use http proxy for some OS bootstrap workflow, they could modify “repo” option to still use `{{ api.server }}` for the address of RackHD on-http service (take [sample payload](#) as an example):

```
...
"install-os": {
  "version": "7.0",
  "repo": "{{ api.server }}/Centos/7.0",
  "rootPassword": "root"
}
...
```

RackHD Web UI

The latest version of the GUI is available publicly at <http://rackhd.github.io/on-web-ui> you can also [download](#) a zip of the latest version.

This zip file can be extracted inside “on-http/static/http” to serve the UI from the MonoRail API server.

Source code for the web user interface is available at <https://github.com/RackHD/on-web-ui>. There is also a [README](#) for learning how to about UI development.

The screenshot shows the RackHD Dashboard interface. At the top, there's a navigation bar with a hamburger menu, a 'Dashboard' label, and a share icon. Below this, the 'Nodes (3)' section is highlighted in blue, with a 'CREATE NODE' button. It contains a table with columns: Name, Type, and Updated. The table lists three nodes: '08.00.27.34-bf-a2' (compute, a month ago), 'Enclosure Node 0' (enclosure, 19 days ago), and '08.00.27.c2-af.43' (compute, 19 days ago). Below the nodes section is the 'Workflows (3)' section, also highlighted in blue, with a 'CREATE WORKFLOW' button. It contains a table with columns: Name, Node, Status, Pending Tasks, Finished Tasks, and Updated. The table lists three workflows: 'Discovery' (Node: 57eb5d, Status: complete, 0 Pending, 10 Finished, 19 days ago), 'Install CentOS' (Node: 282571, Status: complete, 0 Pending, 2 Finished, a month ago), and 'Discovery' (Node: 282571, Status: complete, 0 Pending, 10 Finished, a month ago). At the bottom right, there's an 'EMC²' logo.

Name	Type	Updated
08.00.27.34-bf-a2	compute	a month ago
Enclosure Node 0	enclosure	19 days ago
08.00.27.c2-af.43	compute	19 days ago

Name	Node	Status	Pending Tasks	Finished Tasks	Updated
Discovery	57eb5d	complete	0	10	19 days ago
Install CentOS	282571	complete	0	2	a month ago
Discovery	282571	complete	0	10	a month ago

How to Configure API Endpoint Settings

The screenshot shows a 'Settings' dialog box overlaid on the RackHD interface. A red arrow points from the 'Settings' title to the gear icon in the top right corner of the background dashboard. The dialog box has a title 'Settings' and a label 'MonoRail API Endpoint'. Below the label is a text input field containing the URL 'http://localhost:9090/api/1.1/'. At the bottom of the dialog box are two buttons: 'CANCEL' and 'APPLY'.

1. Once the UI has loaded in your web browser.
2. Click the gear icon located at the top right of the page.
3. Enter the new URL for a running MonoRail API endpoint.
4. Click Apply.

Customize Default iPXE Boot Setting

A compute server's BIOS can be set to always PXE network boot using the BIOS boot order. The default RackHD response when no workflow is operating is to do nothing - normally falling through to the next item in the BIOS boot order. RackHD can also be configured with a default iPXE script to provide boot instructions when no workflow is operational against the node.

Default iPXE Boot Customized OS Into RAM

To configure RackHD to provide a custom iPXE response to a node outside of a workflow running, such as booting a customized kernel and initrd, you can do so by providing configuration to the Node resource in RackHD. This functionality can be enabled by using a PATCH REST API call adding **bootSettings** to a node.

```
curl -X PATCH \
  -H 'Content-Type: application/json' \
  -d @boot.json \
  <server>/api/current/nodes/<identifier>
```

A example of boot.json:

```
{
  "bootSettings": {
    "profile": "defaultboot.ipxe",
    "options": {
      "url": "http://172.31.128.1:9080/common",
      "kernel": "vmlinuz-1.2.0-rancher",
      "initrd": "initrd-1.2.0-rancher",
      "bootargs": "console=tty0 console=ttyS0,115200n8"
    }
  }
}
```

For **bootSettings**, **profile** and **options** are **MUST** required:

Name	Type	Flags	Description
profile	String	required	Profile that will be rendered by RackHD and used by iPXE
options	Object	required	Options in JSON format used to render variables in <i>profile</i>

A default iPXE **profile** *defaultboot.ipxe* is provided by RackHD, and its **options** includes *url*, *kernel*, *initrd*, *bootargs*

Name	Type	Flags	Description
url	String	re- quired	Location Link of <i>kernel</i> and <i>initrd</i> , it could be accessed by http in node, the http service is located in RackHD server or an external server which could be accessed by http proxy or after setting NAT in RackHD. In RackHD server, the root location could be set by <i>httpStaticRoot</i> in <i>config.json</i> or in SKU Pack's <i>config.json</i>
kernel	String	re- quired	Kernel to boot
initrd	String	re- quired	Init ramdisk to boot with <i>kernel</i>
bootargs	String	re- quired	Boot arguments of <i>kernel</i>

Customize iPXE Boot Profile

profile in **bootSettings** could be customized instead of *defaultboot.ipxe*. *defaultboot.ipxe* is provided by default, and its **options** *url*, *kernel*, *initrd*, *bootargs* are aligned with the variables `<%=url%>` `<%=kernel%>` `<%=initrd%>`

`<%=bootargs%>` in *defaultboot.ipxe*, so if the profile is customized, the options also should be aligned with the variables that will be rendered in customized iPXE profile just like *defaultboot.ipxe*

defaultboot.ipxe:

```
kernel <%=url%>/<%=kernel%>
initrd <%=url%>/<%=initrd%>
imgargs <%=kernel%> <%=bootargs%>
boot || prompt --key 0x197e --timeout 2000 Press F12 to investigate || exit shell
```

SSDP/UPnP

RackHD on-http service uses **SSDP** (Simple Service Discovery Protocol) to advertise its Restful API services and device descriptions. The on-http service will respond to M-SEARCH queries from SSDP enabled clients for requested discovery.

Northbound M-SEARCH Queries

- Request all: **ssdp:all**
- Request Root device description: **upnp:rootdevice**
- Request on-http device description: **urn:schemas-upnp-org:device:on-http:1**
- Request API v1.1 service: **urn:schemas-upnp-org:service:api:1.1**
- Request API v2.0 service: **urn:schemas-upnp-org:service:api:2.0**
- Request Redfish v1.0 service: **urn:dmf-org:service:redfish-rest:1.0**
- Example Response:

```
{
  "ST": "urn:dmf-org:service:redfish-rest:1.0",
  "USN": "564d4f6e-a405-706e-38ec-da52ad81e97a:urn:dmf-org:service:redfish-rest:1.0",
  "LOCATION": "http://10.2.3.1:8080/redfish/v1/",
  "CACHE-CONTROL": "max-age=1800",
  "DATE": "Tue, 31 May 2016 18:43:29 GMT",
  "SERVER": "node.js/5.0.0 uPnP/1.1 on-http",
  "EXT": ""
}
```

Southbound M-SEARCH Queries

- Request all: **ssdp:all**
- Request API v1.1 service: **urn:schemas-upnp-org:service:api:1.1:southbound**
- Request API v2.0 service: **urn:schemas-upnp-org:service:api:2.0:southbound**
- Request Redfish v1.0 service: **urn:dmf-org:service:redfish-rest:1.0:southbound**
- Example Response:

```
{
  "ST": "urn:schemas-upnp-org:service:api:2.0:southbound",
  "USN": "564d4f6e-a405-706e-38ec-da52ad81e97a:urn:schemas-upnp-org:service:api:2.0:southbound",
  "LOCATION": "http://172.31.128.1:9080/api/2.0/",
  "CACHE-CONTROL": "max-age=1800",
```



```

"DATE": "Tue, 31 May 2016 18:43:29 GMT",
"SERVER": "node.js/5.0.0 uPnP/1.1 on-http",
"EXT": ""
}

```

Southbound Advertisement Handler

RackHD will poll for SSDP/UPnP advertisements made by nodes residing on the southbound side network. For each advertisement RackHD will publish an alert event to the **on.ssdp** AMQP exchange to notify layers sitting above RackHD.

- Exchange: **on.ssdp**
- Routing Key prefix: **ssdp.alert.***
- AMQP published message example:

```

{
  "delivery_info": {
    "consumer_tag": "None1",
    "delivery_tag": 1734,
    "exchange": "on.ssdp",
    "redelivered": false,
    "routing_key": "ssdp.alert.uuid:f40c2981-7329-40b7-8b04-27f187aecfb5::urn:schemas-upnp-org:s
  },
  "message": {
    "value": {
      "headers": {
        "CACHE-CONTROL": "max-age=1800",
        "DATE": "Mon, 06 Jun 2016 17:09:34 GMT",
        "EXT": "",
        "LOCATION": "172.31.129.47/desc.html",
        "SERVER": "node.js/0.10.25 UPnP/1.1 node-ssdp/2.7.1",
        "ST": "urn:schemas-upnp-org:service:ConnectionManager:1",
        "USN": "uuid:f40c2981-7329-40b7-8b04-27f187aecfb5::urn:schemas-upnp-org:service:Conne
      },
      "info": {
        "address": "172.31.129.47",
        "family": "IPv4",
        "port": 1900,
        "size": 329
      }
    }
  },
  "properties": {
    "content_type": "application/json",
    "type": "Result"
  }
}

```

Configuration Options

Related options defined in *config.json*. For complete examples see [Configuration](#).

Parameter	Description
enableUPnP	boolean true or false to enable or disable all SSDP related server/client services.
ssdpBindAddress	The bind address to send advertisements on (defaults to 0.0.0.0).

UCS-Service

The UCS-Service is an optional RackHD service that will enable RackHD to communicate with Cisco UCS Manager. This allows RackHD to discover and manage the hardware under the UCS Manager.

UCS-Service Setup

The UCS-Service configuration can be set in the config.json file. The following options are supported:

Option	Description
address	IP address the UCS-service will bind to
port	TCP port the UCS-service will bind to
httpsEnabled	set to “true” to enable https access
certFile	Certificate file for https (null for self signed)
keyFile	Key file for https (null for self signed)
debug	set to “true” to enable debugging
callbackUrl	RackHD callback API. ucs-service asynchronous API will post data to RackHD via this callback
concurrency	Celery concurrent process number, default is 2
session	After ucs-service login UCSM, it will keep login active for a duration of “session”, default it is 60 seconds

To start the UCS-Service run:

```
$ pip install -r requirements.txt
$ python app.py
$ python task.py worker
```

Or if you system has supervisord installed, you can use the script ucs-service-ctl.sh to start UCS-service:

```
sudo ./ucs-service-ctl.sh start
```

After you start UCS-service with ucs-service-ctl.sh, you can also stop or restart it with:

```
sudo ./ucs-service-ctl.sh stop/restart
```

There is a supervisord web GUI that can also be used to control ucs-service, by browsing https://<RackHD_Host>:9001

UCS-Service API

The API for the UCS-Service can be accessed via a graphical GUI by directing a browser to https://<RackHD_Host>:7080/ui UCS-service is originally built with synchronous http/https APIs, later on some asynchronous APIs are also developed to improve performance accessing UCSM. UCS-service asynchronous API uses Celery as task queue tool. If user accessed UCS-service asynchronous API, user won't get required data immediately but a response body only includes string “Accepted”. Real data will be posted to **callbackUrl** retrieved from config.json.

UCS-Service Workflows

Default workflows to discover and catalog UCS nodes have been created. There are separate workflows to discover physical UCS nodes, discover logical UCS servers, and to catalog both physical and logical UCS nodes.

Discover Nodes

The Graph.Ucs.Discovery workflow will discover and catalog all physical and logical servers being managed by the specified UCS Manager. It will create a node for each discovered service. It will also create a ucs-obm-service for each node. This obm service can then be used to manage the node. The user must provide the address and login credentials for the UCS manger and the URI for the ucs-service. Below is an example:

```
{
  "name": "Graph.Ucs.Discovery",
  "options":
  {
    "defaults":
    {
      "username": "admin",
      "password": "secret",
      "ucs": "172.31.128.252",
      "uri": "https://localhost:7080"
    },
    "when-discover-physical-ucs":
    {
      "discoverPhysicalServers": "true"
    },
    "when-discover-logical-ucs":
    {
      "discoverLogicalServer": "true"
    },
    "when-catalog-ucs":
    {
      "autoCatalogUcs": "true"
    }
  }
}
```

Field	Description
username	The username used to log into the UCS Manager
password	The password used to log into the UCS Manager
ucs	The hostname or IP address of the UCS Manager
uri	The URI used to access the running UCS-service
discoverPhysicalServers	If set to true, the workflow will create nodes for all physical servers discovered from the UCS Manager
discoverLogicalServer	If set to true, the workflow will create nodes for all logical servers discovered from the UCS Manger
autoCatalogUcs	If set to true, catalog information will be collected for each discovered node

Catalog Nodes

Once the UCS nodes have been discovered, the Graph.Ucs.Catalog can be run with the NodeId. This graph will use the ucs-obm-service created by the discovery workflow so no other options are required.

Tutorials

Overview

This tutorial will show you how to use Docker Compose to run a demo RackHD environment.

The RackHD services will all be run in separate Docker containers. Docker Compose will be used to coordinate running the container and to set up the required networks. A separate Docker Compose file will be used to run virtual compute nodes.

After RackHD is set up successfully, RackHD's discovery, catalog and poller functionality will be introduced by using the simulated nodes that are run in a separate Docker Compose session. In addition, you have the opportunity to experiment with some RackHD APIs. In this module, you will learn about two different RESTful endpoints in RackHD and experiment with them. Additionally, the RackHD set up is used to perform an unattended OS install onto a virtual node.

Finally, if you want to learn more about relevant knowledge, you can go to following links.

- RackHD: <https://github.com/RackHD>
- Docker: <https://www.docker.com/>
- Docker Compose: <https://docs.docker.com/compose/>
- Infrasim: <https://infrasim.readthedocs.io>

RackHD: Local Docker Based Environment Set-up

This tutorial will walk you through getting an instance of RackHD up and running on your local desktop or laptop, this will enable you to see the hosted API documentation and experiment with the APIs.

Prerequisites

jq

You can get details on how to use `jq` at <https://stedolan.github.io/jq/manual/>. By default it colorizes the syntax highlighting and pretty prints javascript data structures.

Another option that can provide the same pretty printing is use:

```
| python -mjson.tool
```

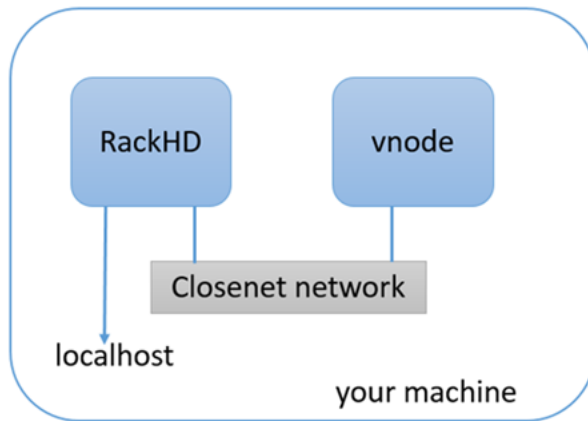
You will need to install [Docker](#) and [Docker Compose](#) before setting up the environment.

If you would try provisioning ESXi on the virtual node, change settings of the kvm module in the host OS.

```
rmmod kvm_intel
rmmod kvm
modprobe kvm ignore_msrs=1
modprobe kvm_intel eptad=1 nested=1
```

You may also want to consider installing `jq` which provides a command-line oriented tool for pretty printing and filtering JSON structured data.

What We're Setting up



The Docker Compose file will download the latest released versions of the RackHD Services from the RackHD DockerHub. It will create two docker bridge networks to run the services. The `rackhd_admin` network will be used to connect the services together and to access the RackHD APIs. The `rackhd_southbound` network will be used by RackHD to connect to the virtual nodes.

The Docker Compose setup also enables port forwarding that allows your localhost to access the RackHD instance:

- localhost:9090 redirects to rackhd_admin:9090 for access to the REST API
- localhost:9093 redirects to rackhd_admin:8443 for secure access to the REST API

Set up RackHD in Docker using Docker Compose

How to set up these environments is shown as follows.

1. Clone the RackHD repository

```
git clone https://github.com/RackHD/RackHD.git
cd RackHD/example/rackhd
```

2. Download and start the RackHD services with the docker-compose up file. This command will load the configuration from the docker-compse.yml file.

```
sudo docker-compose -f docker-compose.yml up -d
```

```

$ |demo-docker 5:1 ?:1 X| → docker-compose up -d
Creating rackhd_mongo_1 ...
Creating rackhd_mongo_1 ... done
Creating rackhd_dhcp_1 ...
Creating rackhd_files_1 ...
Creating rackhd_rabbitmq_1 ...
Creating rackhd_dhcp_1
Creating rackhd_files_1
Creating rackhd_rabbitmq_1 ... done
Creating rackhd_dhcp-proxy_1 ...
Creating rackhd_syslog_1 ...
Creating rackhd_http_1 ...
Creating rackhd_syslog_1
Creating rackhd_dhcp-proxy_1
Creating rackhd_syslog_1 ... done
Creating rackhd_tftp_1 ...
Creating rackhd_http_1 ... done
Creating rackhd_taskgraph_1
Creating rackhd_tftp_1 ... done

```

3. Now use the `docker-compose ps` command to check whether the RackHD services are now up and running.

```
sudo docker-compose ps
```

If RackHD is set up successfully, the result will be shown as follows.

Name	Command	State	Ports
rackhd_dhcp-proxy_1	node /RackHD/on-dhcp-proxy ...	Up	
rackhd_dhcp_1	/docker-entrypoint.sh	Up	
rackhd_files_1	/docker-entrypoint.sh	Up	
rackhd_http_1	node /RackHD/on-http/index.js	Up	
rackhd_mongo_1	docker-entrypoint.sh mongod	Up	27017/tcp, 0.0.0.0:9090->9090/tcp
rackhd_rabbitmq_1	docker-entrypoint.sh rabbit ...	Up	
rackhd_syslog_1	node /RackHD/on-syslog/ind ...	Up	
rackhd_taskgraph_1	node /RackHD/on-taskgraph/ ...	Up	
rackhd_tftp_1	node /RackHD/on-tftp/index.js	Up	

- The command `sudo docker-compose logs` will output the logs from all the running RackHD services. Additionally, you can stop the services with the command `sudo docker-compose stop`, or stop and delete the services with `sudo docker-compose down`.

Automatic Discovery and Catalog Server Nodes

In this module, you will learn about RackHD's discovery, catalog and poller functionality by using the simulated virtual nodes created by Infrsim.

- Discovery: RackHD can automatically discover a node that attempts to do PXE boot on the network that RackHD is monitoring.
- Catalog: RackHD can capture the nodes' attributes and capabilities.
- Poller: RackHD can periodically capture nodes' telemetry data from the hardware interfaces.

Discovery

The `infrsim/default.yml` file (dir: `RackHD/example`) is used to defined the virutal node's configuration. This demo uses an emulated `quanta_d52` vnode. If you want to learn more about InfraSIM, you can go to <https://github.com/InfraSIM>.

UltraVNC Viewer is used to show the console of vnode. In this environment, the vnode console will be availabe on the `rackhd_southbound` network on port `5901`.

1. Start up a vnode

To start a vnode, the follwing command should be run from the `Rackhd/example/infrsim` direcotry.

```
sudo docker-compose up -d
```

You can execute command on host to check whether `quanta_d51` vnode is up successfully. If the status of `quanta_d51` vnode is running, `quanta_d51` is up successfully.

```
sudo docker-compose ps
```

Once the virutual node is up, you can connect to it via a vnc viewer on port `5901`. You first need to look at the log output to find the ip address of the bridge, `br0`, the virtual node is connected to.

```
sudo docker-compose logs
```

```

± |demo-docker S:1 ? :1 X| → docker-compose logs
Attaching to infrasm_infrasm_1
infrasm_1 | mv: cannot move '/etc/resolv.conf.dhclient-new.21' to '/etc/resolv.conf': Device or res
source busy
infrasm_1 | br0      Link encap:Ethernet  HWaddr 02:42:ac:1f:80:03
infrasm_1 |             inet addr:172.31.128.100  Bcast:172.31.143.255  Mask:255.255.240.0
infrasm_1 |             UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
infrasm_1 |             RX packets:5 errors:0 dropped:0 overruns:0 frame:0
infrasm_1 |             TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
infrasm_1 |             collisions:0 txqueuelen:1000
infrasm_1 |             RX bytes:832 (832.0 B)  TX bytes:684 (684.0 B)
infrasm_1 |
infrasm_1 | eth0      Link encap:Ethernet  HWaddr 02:42:ac:1f:80:03
infrasm_1 |             UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
infrasm_1 |             RX packets:6 errors:0 dropped:0 overruns:0 frame:0
infrasm_1 |             TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
infrasm_1 |             collisions:0 txqueuelen:0
infrasm_1 |             RX bytes:992 (992.0 B)  TX bytes:684 (684.0 B)
infrasm_1 |
infrasm_1 | lo        Link encap:Local Loopback
infrasm_1 |             inet addr:127.0.0.1  Mask:255.0.0.0
infrasm_1 |             UP LOOPBACK RUNNING  MTU:65536  Metric:1
infrasm_1 |             RX packets:0 errors:0 dropped:0 overruns:0 frame:0
infrasm_1 |             TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
infrasm_1 |             collisions:0 txqueuelen:1000
infrasm_1 |             RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
infrasm_1 |
infrasm_1 | [ 47      ] default-socat starts to run
infrasm_1 | [ 50      ] default-bmc starts to run
infrasm_1 | [ 60      ] default-node is running
infrasm_1 | Infrasm service started.
infrasm_1 | Node default graphic interface accessible via:
infrasm_1 | VNC port: 5901
infrasm_1 | Either host IP: ['127.0.0.1', '172.31.128.100']
infrasm_1 | depending on host in which network VNC viewer is running

```

You can find the vnode's ip&port is *172.31.128.100:5901*. And you can use the vncviewer to connect this node. There are two scenes. - If your vncviewer tool and docker host are the same local network. You can connect the vnode using vncviewer tool directly. - If not, eg: your docker host is a remote machine with ubuntu system, and the vncviewer is installed in the local windows machine yte vncviewer is installed in the local windows machine. Please follow the three steps:

First, install vnc4server in your docker host (suppose its ip is *1.2.3.4*). eg: install vnc4server in ubuntu 14.04 OS: *sudo apt-get install vnc4server*. Second, run vnodeDiscovery.sh script in your docker host, you can find the mapping port.

```

#/bin/bash -x
HOST_IP=`ifconfig eth0 | grep "inet addr" | awk 'BEGIN{FS=" "}{print $2}' | awk 'BEGIN{FS=":"}{print $1}'`
echo "Host eth0 IP is $HOST_IP"

for i in {2..255}
do
    NUMBER=$i
    IPADDR="172.31.128.$NUMBER"
    PORTNUM=`expr 28000 + $NUMBER`
    sudo iptables -P INPUT ACCEPT
    sudo iptables -P FORWARD ACCEPT
    sudo sysctl net.ipv4.ip_forward=1 > /dev/null
    sudo iptables -A PREROUTING -t nat -p tcp -d $HOST_IP --dport $PORTNUM -j DNAT --to-destination $IPADDR:5901
    sudo iptables -t nat -A POSTROUTING -d $IPADDR -p tcp --dport 5901 -j MASQUERADE
    echo "Setting VNC port $PORTNUM for IP $IPADDR"
done

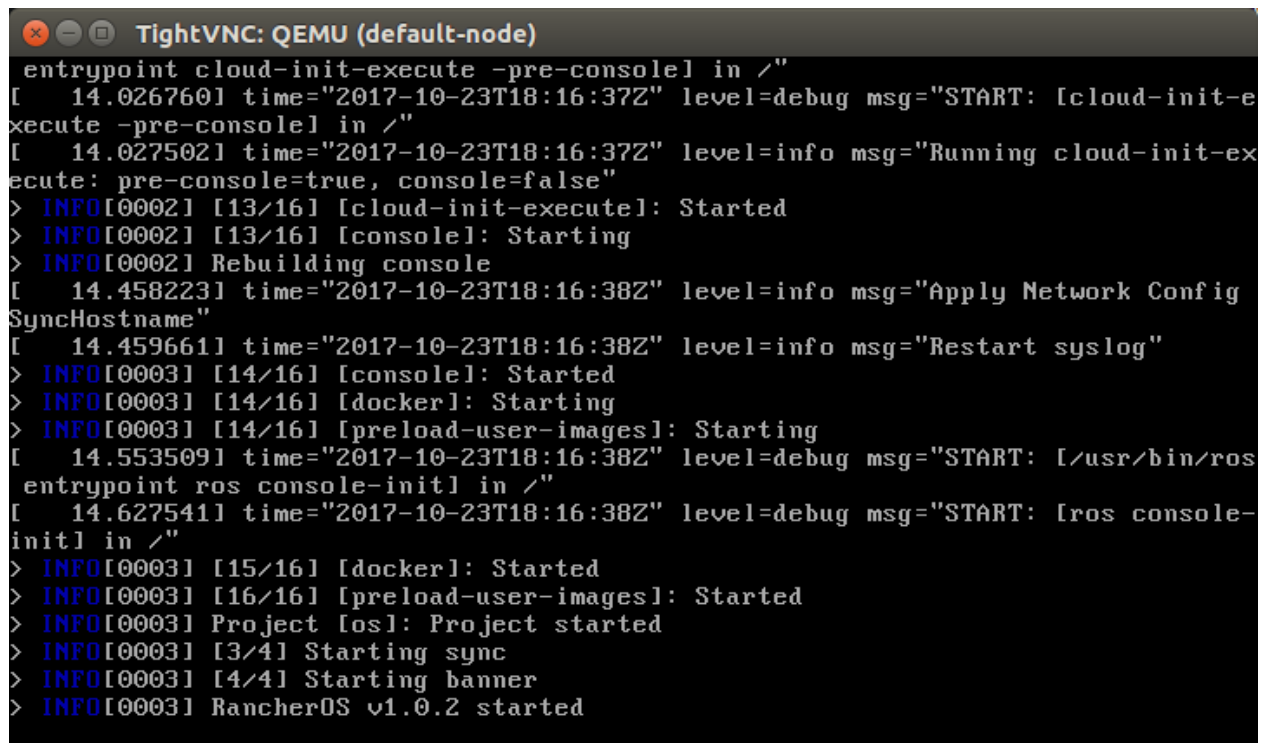
```

```

Setting VNC port 28091 for IP 172.31.128.91
Setting VNC port 28092 for IP 172.31.128.92
Setting VNC port 28093 for IP 172.31.128.93
Setting VNC port 28094 for IP 172.31.128.94
Setting VNC port 28095 for IP 172.31.128.95
Setting VNC port 28096 for IP 172.31.128.96
Setting VNC port 28097 for IP 172.31.128.97
Setting VNC port 28098 for IP 172.31.128.98
Setting VNC port 28099 for IP 172.31.128.99
Setting VNC port 28100 for IP 172.31.128.100
Setting VNC port 28101 for IP 172.31.128.101
Setting VNC port 28102 for IP 172.31.128.102
Setting VNC port 28103 for IP 172.31.128.103
Setting VNC port 28104 for IP 172.31.128.104
Setting VNC port 28105 for IP 172.31.128.105
Setting VNC port 28106 for IP 172.31.128.106

```

Third, use vncviewer tool to connect the vnode using the vnc server address `1.2.3.4:28100`. Vncviewer download address: <http://www.uvnc.com/downloads/ultravnc.html>

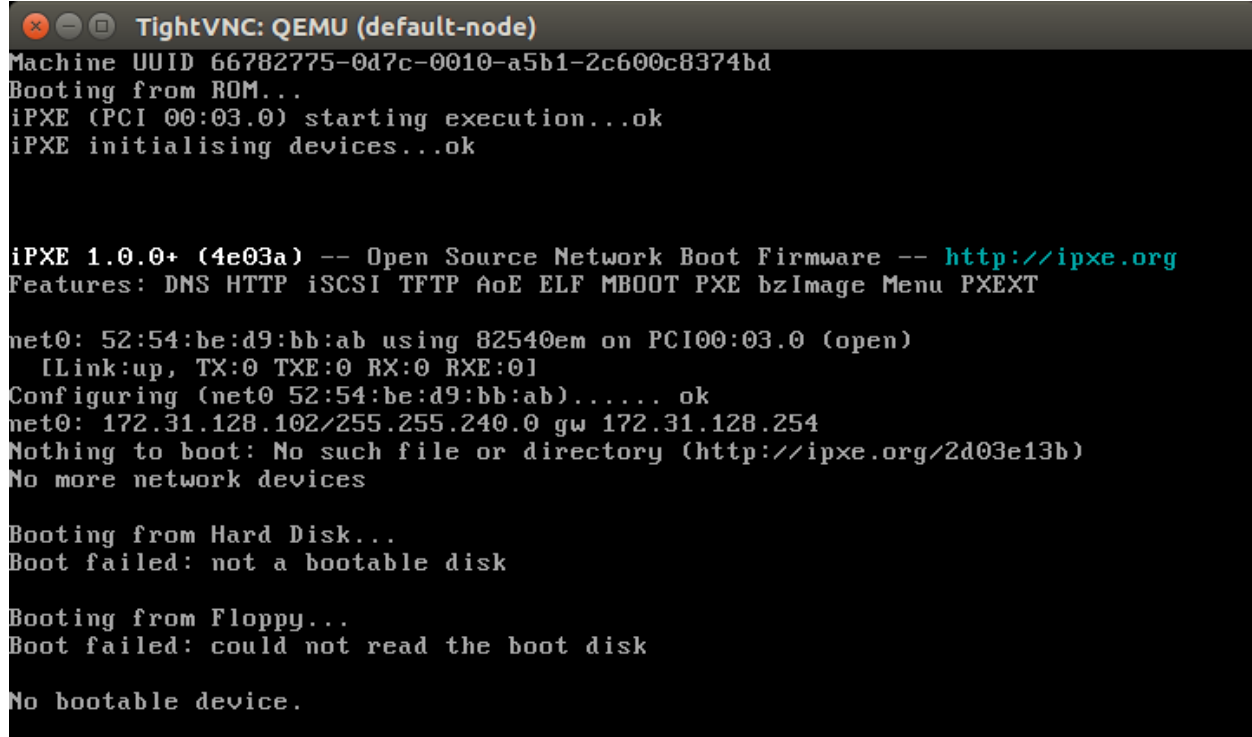


```

TightVNC: QEMU (default-node)
entrypoint cloud-init-execute -pre-console] in /"
[ 14.026760] time="2017-10-23T18:16:37Z" level=debug msg="START: [cloud-init-e
xecute -pre-console] in /"
[ 14.027502] time="2017-10-23T18:16:37Z" level=info msg="Running cloud-init-ex
ecute: pre-console=true, console=false"
> INFO[0002] [13/16] [cloud-init-execute]: Started
> INFO[0002] [13/16] [console]: Starting
> INFO[0002] Rebuilding console
[ 14.458223] time="2017-10-23T18:16:38Z" level=info msg="Apply Network Config
SyncHostname"
[ 14.459661] time="2017-10-23T18:16:38Z" level=info msg="Restart syslog"
> INFO[0003] [14/16] [console]: Started
> INFO[0003] [14/16] [docker]: Starting
> INFO[0003] [14/16] [preload-user-images]: Starting
[ 14.553509] time="2017-10-23T18:16:38Z" level=debug msg="START: [/usr/bin/ros
entrypoint ros console-init] in /"
[ 14.627541] time="2017-10-23T18:16:38Z" level=debug msg="START: [ros console-
init] in /"
> INFO[0003] [15/16] [docker]: Started
> INFO[0003] [16/16] [preload-user-images]: Started
> INFO[0003] Project [os]: Project started
> INFO[0003] [3/4] Starting sync
> INFO[0003] [4/4] Starting banner
> INFO[0003] RancherOS v1.0.2 started

```

3. The vNode console will pause for about 1 minute while Rackhd catalogs the vnode information. The vNode will reboot after cataloging finishes. This reboot indicates that the discovery workflow is completed.



```
TightVNC: QEMU (default-node)
Machine UUID 66782775-0d7c-0010-a5b1-2c600c8374bd
Booting from ROM...
iPXE (PCI 00:03.0) starting execution...ok
iPXE initialising devices...ok

iPXE 1.0.0+ (4e03a) -- Open Source Network Boot Firmware -- http://ipxe.org
Features: DNS HTTP iSCSI TFTP AoE ELF MBOOT PXE bzImage Menu PXEXT

net0: 52:54:be:d9:bb:ab using 82540em on PCI00:03.0 (open)
  [Link:up, TX:0 TXE:0 RX:0 RXE:0]
Configuring (net0 52:54:be:d9:bb:ab)..... ok
net0: 172.31.128.102/255.255.240.0 gw 172.31.128.254
Nothing to boot: No such file or directory (http://ipxe.org/2d03e13b)
No more network devices

Booting from Hard Disk...
Boot failed: not a bootable disk

Booting from Floppy...
Boot failed: could not read the boot disk

No bootable device.
```

4.Retrieve the nodes by typing the following RackHD API to discover the node.

```
curl localhost:9090/api/2.0/nodes | jq '.'
```

You can see one or more vnodes whose type is “enclosure” or “compute”.

```

{
  "autoDiscover": "false",
  "catalogs": "/api/2.0/nodes/58b66105a36ced790cd0108c/catalogs",
  "id": "58b66105a36ced790cd0108c",
  "identifiers": [],
  "name": "Enclosure Node 18F2182",
  "obms": [],
  "tags": "/api/2.0/nodes/58b66105a36ced790cd0108c/tags",
  "pollers": "/api/2.0/nodes/58b66105a36ced790cd0108c/pollers",
  "relations": [
    {
      "relationType": "encloses",
      "info": ,
      "targets": [
        "58b660116d20657f0c5d6466"
      ]
    }
  ],
  "type": "enclosure",
  "workflows": "/api/2.0/nodes/58b66105a36ced790cd0108c/workflows"
},
{
  "autoDiscover": "false",
  "catalogs": "/api/2.0/nodes/58b660116d20657f0c5d6466/catalogs",
  "id": "58b660116d20657f0c5d6466",
  "identifiers": [
    "00:60:16:6d:4e:c4"
  ],
  "name": "00:60:16:6d:4e:c4",
  "obms": [],
  "tags": "/api/2.0/nodes/58b660116d20657f0c5d6466/tags",
  "pollers": "/api/2.0/nodes/58b660116d20657f0c5d6466/pollers",
  "relations": [
    {
      "relationType": "enclosedBy",
      "info": ,
      "targets": [
        "58b66105a36ced790cd0108c"
      ]
    }
  ],
  "sku": ,
  "type": "compute",
  "workflows": "/api/2.0/nodes/58b660116d20657f0c5d6466/workflows"
}

```

NodeId

NodeId is the unique Identity of a node in RackHD. List all the compute type nodes being discovered on the rackhd-server SSH console by typing the following command. Append "?type=compute" as a query string.

You will focus on compute type nodes in this module

```
curl 127.0.0.1:9090/api/2.0/nodes?type=compute | jq '.'
```

In the following json output, the compute node ID is 58b660116d20657f0c5d6466. You will take it as a variable named <node_id> in the following module.

Note: The node_id varies for different nodes. Even for the same node, the Node ID changes if the RackHD database is being cleaned and the node rediscovered.

Do not use the example 58b660116d20657f0c5d6466 in your vLab. Use the displayed Node ID in your lab.

Retrieve Catalogs

Catalogs are described as the following:

- Free form data structures with information about the nodes
- Pluggable mechanisms for adding new catalogers for additional data

- JSON documents stored in MongoDB

Examples of catalog sources include the following:

- Drive smart information
- DriveId catalog including system identified drive information
- DMI from *dmidecode* command
- OHAI aggregate of different stats in more friendly JSON format
- IPMI information gets per ipmitool over KCS channel LAN information
- FRU, SEL, SDR, MC information
- *lsscsi*, *lspci*, *lshw* commands output
- Raid information can be got via storcli/perccli tool
- Dell computers provide some catalogs retrieved from RACADM tool
- LLDP

Specify The Catalogs Source

1. To view the sources where the catalogs data was retrieved from, type the following command.

Note: the <node_id> is the Node-ID retrieved from Step 3.

```
curl 127.0.0.1:9090/api/2.0/nodes/<node_id>/catalogs/ | jq '.' | grep source
```

```
"source": "bmc",
"source": "ipmi-sel-information",
"source": "ipmi-sel",
"source": "ipmi-mc-info",
"source": "ipmi-user-summary-1",
"source": "dmi",
"source": "ohai",
"source": "ipmi-user-list-1",
"source": "ipmi-fru",
"source": "ipmi-user-summary-2",
"source": "ipmi-user-list-2",
"source": "rmm-user-summary",
"source": "rmm-user-list",
"source": "ipmi-user-summary-4",
"source": "ipmi-user-list-4",
"source": "ipmi-user-summary-5",
"source": "ipmi-user-list-5",
"source": "ipmi-user-summary-6",
"source": "ipmi-user-list-6",
"source": "ipmi-user-summary-7",
"source": "ipmi-user-list-7",
"source": "ipmi-user-summary-8",
```

2. Select one of the sources you are interested in, and then append to the command. For example, the following example use ipmi-fru

```
curl 127.0.0.1:9090/api/2.0/nodes/<node_id>/catalogs/ipmi-fru | jq '.'
```

or “driveId” as example

```
curl 127.0.0.1:9090/api/2.0/nodes/<node_id>/catalogs/driveId | jq '.'
```

Retrieve Pollers

What's Poller

- The pollers API provides functionality for periodic collection of IPMI and SNMP data.
- IPMI Pollers can be standalone or can be associated with a node. When an IPMI poller is associated with a node, it will attempt to use that node's IPMI OBM settings in order to communicate with the BMC. Otherwise, the poller must be manually configured with that node's IPMI settings.
- SNMP pollers can be standalone or associated with a node. When an SNMP poller is associated with a node, it attempts to use that node's snmpSettings in order to communicate via SNMP. Otherwise, the poller must be manually configured with that node's SNMP settings.

Examples of Telemetry

- Switches Switch CPU, Memory
- Port status
- Port utilization
- Arbitrary MIB gathering capable
- PDU Socket status
- Arbitrary MIB gathering capable
- IPMI Sensors (SDR)
- Power status

Retrieve Pollers

1. On rackhd-server, list the active pollers which by default run in the background, by typing the following command.

```
curl 127.0.0.1:9090/api/2.0/pollers| jq '.'
```

Below is a definition of each field in the example output below:

- “id” is the poller's id. Denote it as <poller_id>. you will refer to later.
- “type” means it is an IPMI poller or SNMP poller, and so on.
- “pollInterval” is the interval for the frequency that RackHD polls that data. The time is the milliseconds to wait between polls.
- “node” is the target node of the poller that the data comes from.
- “command” is the kind of IPMI command that this poller is issued.

Note: Record listed below is an example. The output on your screen will look similar with different data.

```
{
  "id": "58b66105a36ced790cd01091",
  "type": "ipmi",
  "pollInterval": 30000,
  "node": "/api/2.0/nodes/58b660116d20657f0c5d6466",
  "config": {
    "command": "sdr"
  },
  "lastStarted": "2017-03-01T06:22:35.417Z",
```

```
{
  "lastFinished": "2017-03-01T06:22:55.241Z",
  "paused": false,
  "failureCount": 0
}
```

2. Show the poller data, by typing the following command.

```
curl 127.0.0.1:9090/api/2.0/pollers/<poller_id>/data | jq '.'
```

3. Change the interval of a poller, by typing the following command.

```
curl -X PATCH -H 'Content-Type: application/json' -d '{"pollInterval":15000}' 127.0.0.1:9090/api/2.0/pollers/<poller_id>
```

```
[
  {
    "id": "58df2a1c83d44813084b4600",
    "type": "ipmi",
    "pollInterval": 60000,
    "node": "/api/2.0/nodes/58df293a65b2761908e6b78d",
    "config": {
      "command": "sel"
    },
    "lastStarted": "2017-04-01T04:28:59.586Z",
    "lastFinished": "2017-04-01T04:28:59.663Z",
    "paused": false,
    "failureCount": 0
  },
  {
    "id": "58df2a1c83d44813084b4601",
    "type": "ipmi",
    "pollInterval": 60000,
    "node": "/api/2.0/nodes/58df293a65b2761908e6b78d",
    "config": {
      "command": "selInformation"
    },
    "lastStarted": "2017-04-01T04:28:59.578Z",
    "lastFinished": "2017-04-01T04:28:59.641Z",
    "paused": false,
    "failureCount": 0
  },
  {
    "id": "58df2a1c83d44813084b4602",
    "type": "ipmi",
    "pollInterval": 60000,
    "node": "/api/2.0/nodes/58df293a65b2761908e6b78d",
    "config": {
      "command": "selInformation"
    },
    "lastStarted": "2017-04-01T04:28:59.578Z",
    "lastFinished": "2017-04-01T04:28:59.641Z",
    "paused": false,
    "failureCount": 0
  }
]
```

Interfacing with The RESTful API

Overview

In the previous modules, you had the opportunity to experiment with some RackHD APIs. In this module you will learn about two different RESTful endpoints in RackHD and experiment with them.

RackHD is designed to provide a REST (Representational state transfer) architecture to provide a RESTful API. RackHD currently has two RESTful interfaces: a Redfish API and native REST API 2.0.

The Redfish API is compliant with the Redfish specification as an additional REST API. It provides a common data model for representing bare metal hardware, as an aggregate for multiple backend servers and systems.

The REST API 2.0 provides unique features that are not provided in the Redfish API.

Restful API (v2.0)

REST API (v 2.0) - Get workflow history

The <Node-ID> is retrieved from 7.2.

```
curl localhost:9090/api/2.0/nodes/<Node-ID>/workflows | jq '.'
```

```
[
  {
    "node": "58df293a65b2761908e6b78d",
    "definition": {
      "friendlyName": "Discovery",
      "injectableName": "Graph.Discovery",
      "options": {
        "bootstrap-ubuntu": {
          "triggerGroup": "bootstrap"
        },
        "finish-bootstrap-trigger": {
          "triggerGroup": "bootstrap"
        },
        "skip-reboot-post-discovery": {
          "skipReboot": true,
          "when": "({options.skipReboot})"
        },
        "target": "58df293a65b2761908e6b78d",
        "instanceId": "ba62c08c-d7ff-4433-8557-1e834f61c1bb"
      },
      "tasks": [
        {
          "label": "bootstrap-ubuntu",
          "taskName": "Task.Linux.Bootstrap.Ubuntu"
        },
        {
          "label": "catalog-dmi",
```

REST API (v 2.0) - Get active workflow

```
curl localhost:9090/api/2.0/nodes/<Node-ID>/workflows?active=true | jq '.'
```

In the following example, the return is blank ([]), which means no workflow is actively running on this node.

```
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                               Dload  Upload   Total   Spent    Left   Speed
100      2    100      2      0      0     47      0  --:--:--  --:--:--  --:--:--    47
[]
rackhd@rackhd-server:~$
```

REST API (v 2.0) - Show RackHD configurations

Show the RackHD configurations, by running the following command.

```
curl localhost:9090/api/2.0/config | jq '.'
```

```
{
  "amqp": "amqp://localhost",
  "apiServerAddress": "172.31.128.1",
  "apiServerPort": 9080,
  "dhcpGateway": "172.31.128.1",
  "dhcpProxyBindAddress": "172.31.128.1",
  "dhcpProxyBindPort": 4011,
  "dhcpSubnetMask": "255.255.240.0",
  "autoCreateOdm": true,
  "httpEndpoints": [
    {
      "address": "0.0.0.0",
      "port": 8080,
      "httpsEnabled": false,
      "proxiesEnabled": true,
      "authEnabled": false,
      "routers": "northbound-api-router"
    },
    {
      "address": "0.0.0.0",
      "port": 8443,
      "httpsEnabled": true,
      "proxiesEnabled": true,
      "authEnabled": true,
      "routers": "northbound-api-router"
    }
  ],
```

REST API (v 2.0) - lookup table

Dump the IP address in the lookup table (where RackHD maintain the nodes IP), by running the following command.

```
curl localhost:9090/api/2.0/lookups | jq '.'
```

```
{
  "macAddress": "00:50:56:01:34:52",
  "ipAddress": "192.168.1.250",
  "updatedAt": "2017-04-01T04:18:35.378Z",
  "createdAt": "2017-04-01T04:09:09.113Z",
  "id": "58df17e56bf8bd9c4df17116"
},
{
  "node": "58df193a65b2761908e6b78d",
  "macAddress": "00:60:16:bb:eb:7d",
  "ipAddress": "172.31.128.34",
  "updatedAt": "2017-04-01T04:18:35.378Z",
  "createdAt": "2017-04-01T04:09:09.114Z",
  "id": "58df17e56bf8bd9c4df17117"
},
{
  "macAddress": "00:50:56:be:5e:4e",
  "ipAddress": "172.31.128.30",
  "updatedAt": "2017-04-01T04:18:35.379Z",
  "createdAt": "2017-04-01T04:09:09.113Z",
  "id": "58df17e56bf8bd9c4df17118"
}
```

REST API (v 2.0) - built-in workflow

- Show the name of all built-in workflow

```
curl localhost:9090/api/2.0/workflows/graphs | jq '.' | grep injectableName | grep "Graph.*" | grep -v "Graph.*"
```

As below output example, you will find lots of handy built-in workflow which RackHD carries, which you can leverage them directly.

```
"injectableName": "Graph.Add.Volume",
"injectableName": "Graph.Bootstrap.With.BMC.Credentials.Remove",
"injectableName": "Graph.Add.Hotspare",
"injectableName": "Graph.Bootstrap.With.BMC.Credentials.Set",
"injectableName": "Graph.Switch.Discovery.Brocade.Ztp",
"injectableName": "Graph.Raid.Create.MegaRAID",
"injectableName": "Graph.Raid.Delete.MegaRAID",
"injectableName": "Graph.Dell.Disable.VTx",
"injectableName": "Graph.Switch.Discovery.Cisco.Poap",
"injectableName": "Graph.Redfish.Chassis.Poller.Create",
"injectableName": "Graph.Delete.Volume",
"injectableName": "Graph.ClearSEL.Node",
"injectableName": "Graph.Bootstrap.Decommission.Node",
"injectableName": "Graph.Switch.Discovery.Arista.Ztp",
"injectableName": "Graph.Bootstrap.Decommission.Node.Test",
"injectableName": "Graph.Redfish.Managers.Poller.Create",
```

REST API (v 2.0) - issue a workflow

Post a workflow to a specific node by running the following command.

In the following example, to post a workflow to Reset a Node, the Node_id is obtained by the “curl localhost:9090/api/2.0/nodes | jq ‘.’ ” API.

```
curl -X POST -H 'Content-Type: application/json' 127.0.0.1:9090/api/2.0/nodes/<Node_id>/workflows?name=Graph.Reset.Node
```

Then the vNode is powered cycle and rebooted.

```
{
  "definition": {
    "friendlyName": "Reset Node",
    "injectableName": "Graph.Reset.Node",
    "tasks": [
      {
        "label": "reset",
        "taskName": "Task.Obm.Node.Reset"
      }
    ],
    "options": {}
  },
  "instanceId": "c08c0b5a-d6d5-4a71-8fac-f65d15a7b588",
  "context": {
    "target": "58df293a65b2761908e6b78d",
    "graphId": "c08c0b5a-d6d5-4a71-8fac-f65d15a7b588"
  },
  "domain": "default",
  "name": "Reset Node",
  "injectableName": "Graph.Reset.Node",
  "tasks": {
    "a6589e83-473c-499e-9cdf-6f3201e778f1": {
      "friendlyName": "Reset Node",
      "injectableName": "Task.Obm.Node.Reset",
      "implementsTask": "Task.Base.Obm.Node",
      "options": {
        "action": "reset",

```

Redfish API - Chassis

List the Chassis that is managed by RackHD (equivalent to the enclosure node in REST API 2.0), by running the following command.

```
curl 127.0.0.1:9090/redfish/v1/Chassis | jq '.'
```

```
{
  "@odata.context": "/redfish/v1/$metadata#Systems",
  "@odata.id": "/redfish/v1/Chassis",
  "@odata.type": "#ChassisCollection.ChassisCollection",
  "Oem": {},
  "Name": "Chassis Collection",
  "Members@odata.count": 1,
  "Members": [
    {
      "@odata.id": "/redfish/v1/Chassis/58df2a1b83d44813084b45f9"
    }
  ]
}
rackhd@rackhd-server:~$
```

Redfish API - System

1. In the rackhd-server, list the System that is managed by RackHD (equivalent to compute node in API 2.0), by running the following command

```
curl 127.0.0.1:9090/redfish/v1/Systems | jq '.'
```

2. Use the mouse to select the **System-ID** as below example, then the ID will be in your clipboard. This ID will be used in the following steps.

```
{
  "@odata.context": "/redfish/v1/$metadata#Systems",
  "@odata.id": "/redfish/v1/Systems",
  "@odata.type": "#ComputerSystemCollection.ComputerSystemCollection",
  "Oem": {},
  "Name": "Computer System Collection",
  "Members@odata.count": 1,
  "Members": [
    {
      "@odata.id": "/redfish/v1/Systems/58df293a65b2761908e6b78d"
    }
  ]
}
rackhd@rackhd-server:~$
```


Redfish API - SEL Log

```
curl 127.0.0.1:9090/redfish/v1/systems/<System-ID>/LogServices/Sel| jq '.'
```

```
{
  "@odata.context": "/redfish/v1/$metadata#Systems/Links/Members/58df293a65b2761908e6b78d/LogServices/
Members/$entity",
  "@odata.id": "/redfish/v1/systems/58df293a65b2761908e6b78d/LogServices/Sel",
  "@odata.type": "#LogService.1.0.0.LogService",
  "Oem": {},
  "Id": "SEL",
  "Description": "IPMI System Event Log",
  "Name": "ipmi-sel-information",
  "ServiceEnabled": true,
  "MaxNumberOfRecords": 0,
  "OverWritePolicy": "WrapsWhenFull",
  "DateTimeLocalOffset": "+00:00",
  "Actions": {
    "Oem": {},
    "#LogService.ClearLog": {
      "target": "/api/current/node/58df293a65b2761908e6b78d/workflows?name=Graph.ClearSEL.Node"
    }
  },
  "Status": {},
  "Entries": {
    "@odata.id": "/redfish/v1/systems/58df293a65b2761908e6b78d/LogServices/Sel/Entries"
  }
}
```

Redfish API - CPU info

```
curl 127.0.0.1:9090/redfish/v1/Systems/<System-ID>/Processors/0| jq '.'
```

```
{
  "@odata.context": "/redfish/v1/$metadata#Systems/Links/Members/58df293a65b2761908e6b78d/Processors/M
embers/$entity",
  "@odata.id": "/redfish/v1/systems/58df293a65b2761908e6b78d/Processors/0",
  "@odata.type": "#Processor.1.0.0.Processor",
  "Oem": {},
  "Id": "0",
  "Name": "",
  "Socket": "SOCKET 0",
  "ProcessorType": "CPU",
  "ProcessorArchitecture": "x86",
  "InstructionSet": "x86-64",
  "Manufacturer": "Intel",
  "Model": "Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz",
  "MaxSpeedMHz": 2300,
  "TotalCores": 10,
  "TotalThreads": 20,
  "Status": {},
  "ProcessorId": {
    "VendorId": "GenuineIntel",
    "IdentificationRegisters": "F2 06 03 00 FF FB EB BF",
    "EffectiveFamily": "Xeon",
    "EffectiveModel": "Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz",
    "Step": "",
    "MicrocodeInfo": ""
  }
}
```

Redfish API - Helper

Show the list of RackHD Redfish APIs' by running below command:

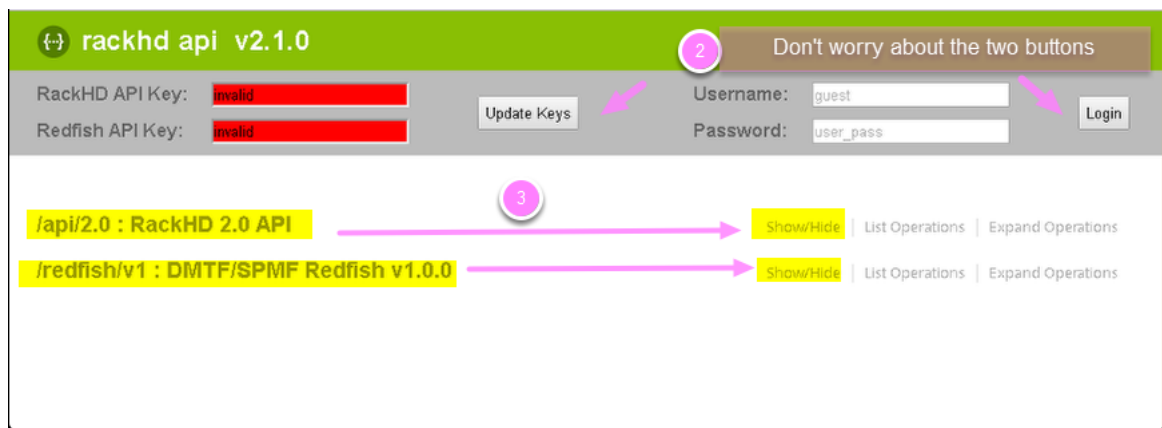
```
curl 127.0.0.1:9090/redfish/v1| jq '.'
```

```
{
  "@odata.context": "/redfish/v1/$metadata#ServiceRoot",
  "@odata.id": "/redfish/v1",
  "@odata.type": "#ServiceRoot.1.0.0.ServiceRoot",
  "Oem": {},
  "Id": "",
  "Description": "",
  "Name": "Root Service",
  "RedfishVersion": "1.0.0",
  "UUID": "423c839f-f5e7-4081-1dbb-ac59ed46267f",
  "Links": {
    "Oem": {},
    "Sessions": {}
  },
  "Systems": {
    "@odata.id": "/redfish/v1/Systems"
  },
  "Chassis": {
    "@odata.id": "/redfish/v1/Chassis"
  },
  "Managers": {
    "@odata.id": "/redfish/v1/Managers"
  },
  "Tasks": {
    "@odata.id": "/redfish/v1/TaskService"
  },
  "SessionService": {
```

Swagger UI

The following steps provide an overview of how to use the Swagger UI.

1. Open “Chrome” and then type `http://<IP>:9090/swagger-ui` in the URL address box. The <IP> should be set to the node ip where the environment is set up. For example, if you set up environment on windows, the <IP> should be set to localhost. If you set up environment on Linux, you can get node ip first by the command `ifconfig eth0` and then replace <IP>.
2. You can ignore the Update Keys button and Login button. They take no effect when “authEnabled == false” is set in the RackHD configuration file (`/opt/monorail/config.json`).
3. Click `/api/2.0` or `/redfish/v1`, to expand the API list.



4. Get a Node ID by scrolling down the /API 2.0 list and clicking the “/nodes” API.
5. Click Get a list of nodes on the right side of the bar, to expand the details for this API. The details for this API are displayed (for example, description, parameters, response code).
6. Click Try it out! button, to invoke this API.

Notes: Some APIs do not require parameters. Some APIs require that you specify a unique Node ID or other parameters. Enter any necessary parameters and Swagger UI builds the RESTful API request and sends it when you click

Try it Out.

GET /nodes Get a list of nodes

Implementation Notes
Get a list of all currently stored nodes.

Response Class (Status 200)
Successfully retrieved the list of nodes

Model | Example Value

```
[
  {}
]
```

Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
\$skip	<input type="text"/>	Query string specifying properties to search for	query	integer
\$top	<input type="text"/>	Query string specifying properties to search for	query	integer
sort	<input type="text"/>	Query string specifying properties to sort with	query	string

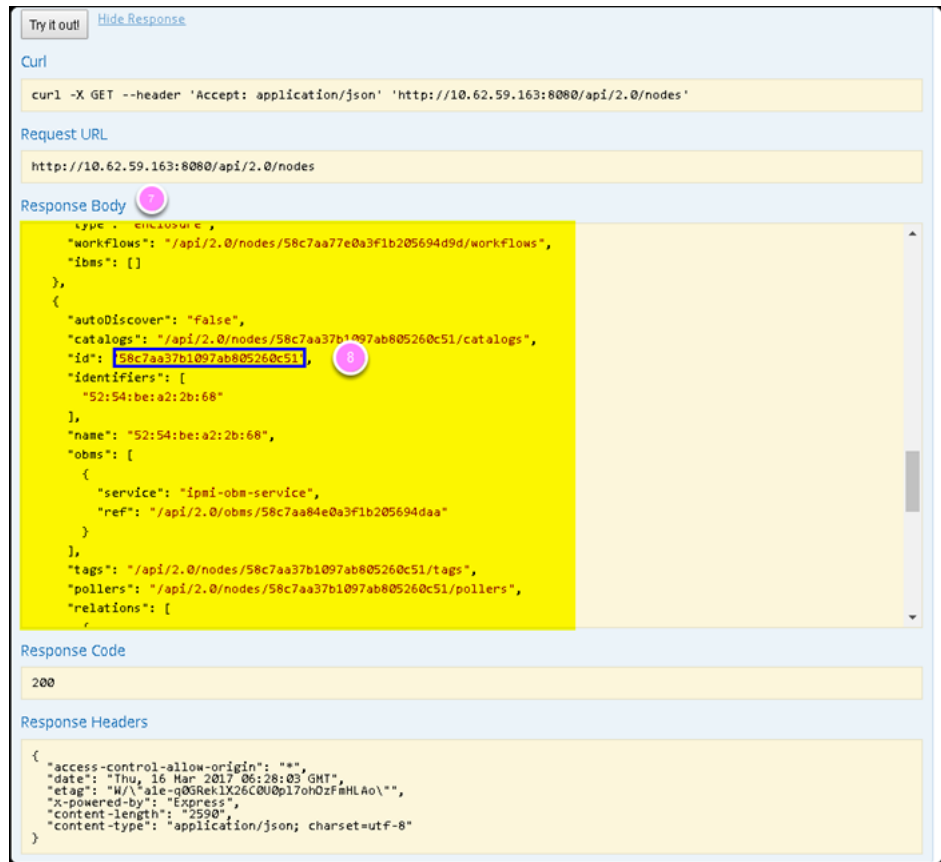
Response Messages

HTTP Status Code	Reason	Response Model	Headers
400	Bad Request	Model Example Value	
		<pre>{ "code": 0, "fields": "string", "message": "string" }</pre>	
default	Unexpected error	Model Example Value	
		<pre>{ "code": 0, "fields": "string", "message": "string" }</pre>	

6

Try it out!

- The RackHD response is displayed. The Response Body is output in a json format from RackHD and is exactly the same as the return from command line API.
- Copy a node ID with the type “compute”, instead of “enclosure”.



Try it out! [Hide Response](#)

Curl

```
curl -X GET --header 'Accept: application/json' 'http://10.62.59.163:8080/api/2.0/nodes'
```

Request URL

<http://10.62.59.163:8080/api/2.0/nodes>

Response Body

```
{
  "type": "BMC-IPMI",
  "workflows": "/api/2.0/nodes/58c7aa37b1097ab805260c51/workflows",
  "ibms": []
},
{
  "autoDiscover": "false",
  "catalogs": "/api/2.0/nodes/58c7aa37b1097ab805260c51/catalogs",
  "id": "58c7aa37b1097ab805260c51",
  "identifiers": [
    "52:54:be:a2:2b:68"
  ],
  "name": "52:54:be:a2:2b:68",
  "obms": [
    {
      "service": "ipmi-obm-service",
      "ref": "/api/2.0/obms/58c7aa84e0a3f1b205694daa"
    }
  ],
  "tags": "/api/2.0/nodes/58c7aa37b1097ab805260c51/tags",
  "pollers": "/api/2.0/nodes/58c7aa37b1097ab805260c51/pollers",
  "relations": [
  ]
}
```

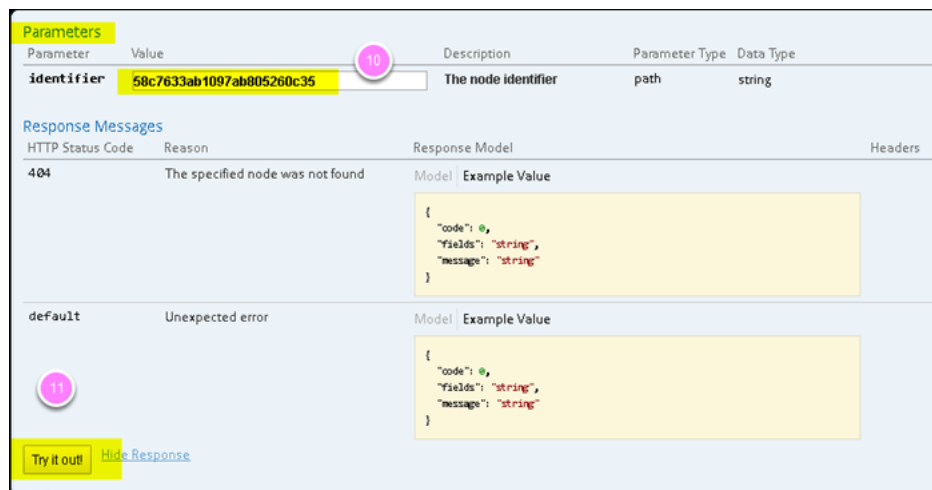
Response Code

200

Response Headers

```
{
  "access-control-allow-origin": "*",
  "date": "Thu, 16 Mar 2017 06:28:03 GMT",
  "etag": "W/\"ale-q05Rek1X26C0U0p17oh0ZfMHLAo\"",
  "x-powered-by": "Express",
  "content-length": "2590",
  "content-type": "application/json; charset=utf-8"
}
```

9. From the API list, under /nodes API, find /nodes/{identifier}/catalogs, and then click the “Get the catalogs from a node”.
10. Paste the node ID that you copied in step 6.
11. Click the **Try it out!**. The catalog data for this node is displayed.
12. Experiment with other APIs.



Parameters

Parameter	Value	Description	Parameter Type	Data Type
identifier	58c7633ab1097ab805260c35	The node identifier	path	string

Response Messages

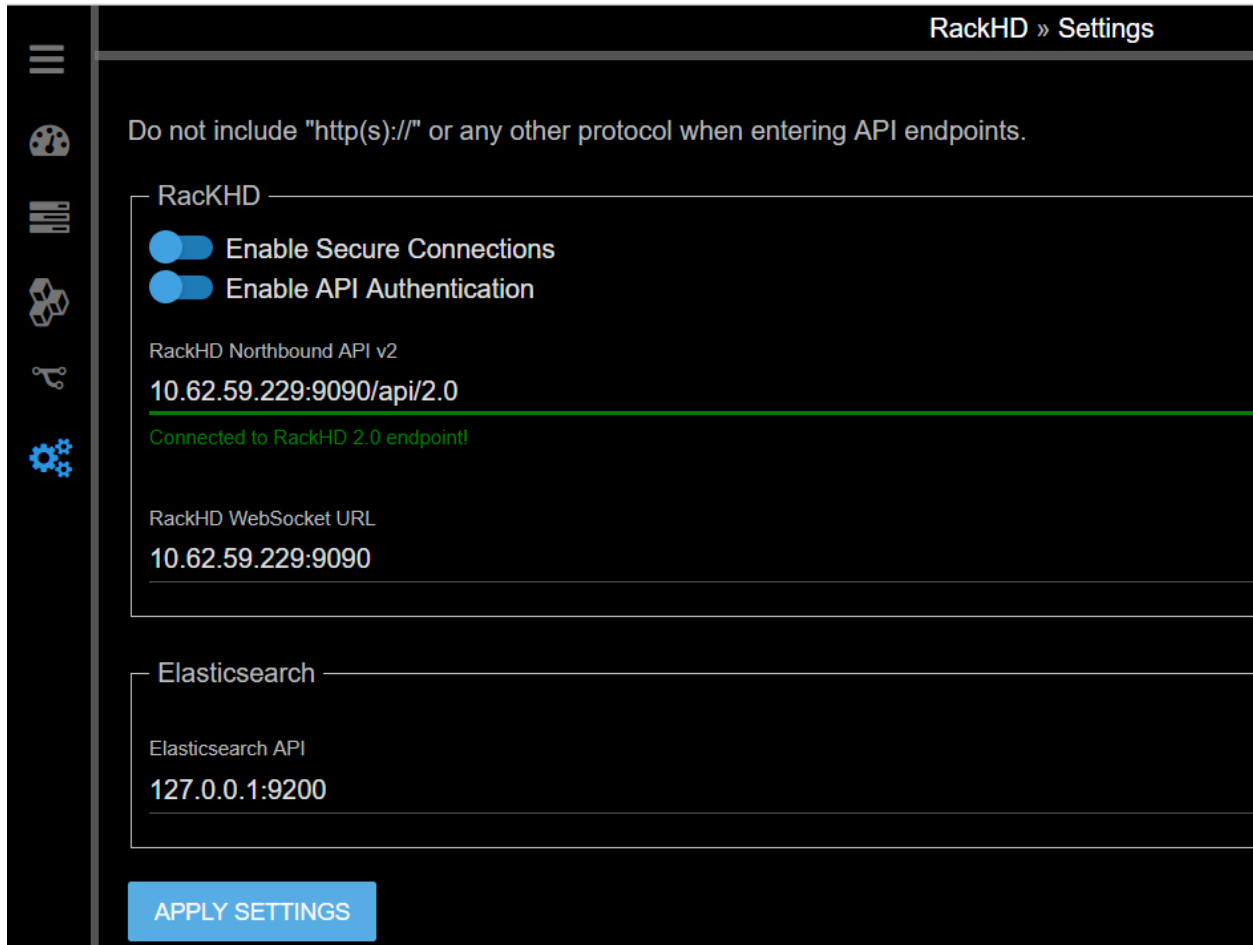
HTTP Status Code	Reason	Response Model	Headers
404	The specified node was not found	Model Example Value	
default	Unexpected error	Model Example Value	

Try it out! [Hide Response](#)

Workflow Editor

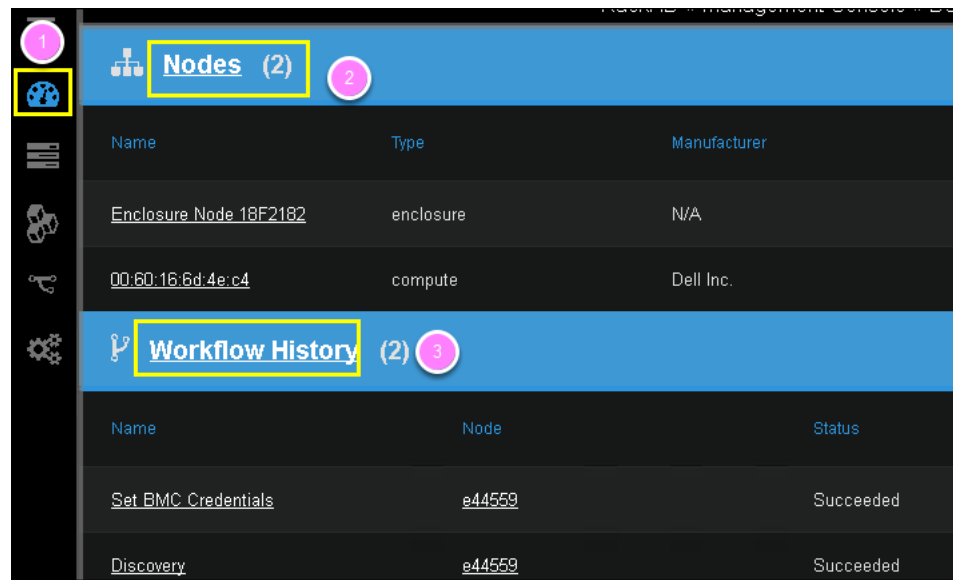
Step 1: Configure on-web-ui

1. On the Windows desktop of launchpad, open Chrome, and then go to the following URL.
`http://localhost:9090/ui`
2. click the “gear” button on the left panel
3. leave the 2 switches as default (in this Lab, in the `/opt/monorail/config.json`, the https is enabled and api authentication is disabled)
4. On the Windows desktop of launchpad, verify the API 2.0 end point (`127.0.0.1:9090/api/2.0`)
5. click “apply settings” button on the bottom.

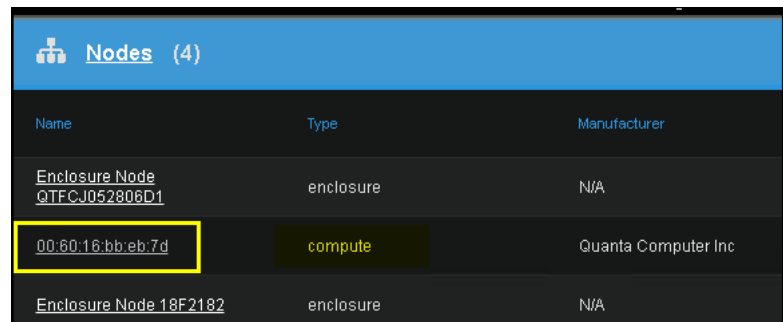


Step 2: Try on-web-ui

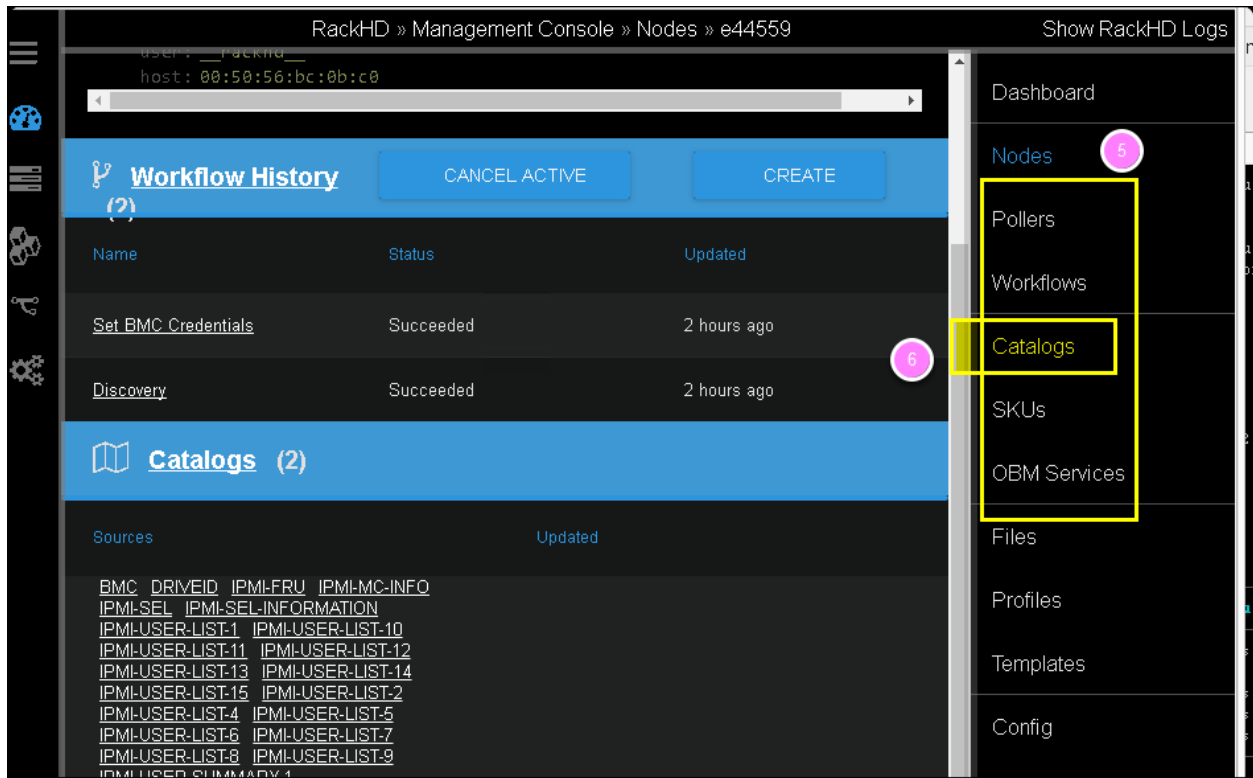
1. Click the meter button in the left panel.
2. You can view the nodes list in the table.
3. You can view the workflow history in the table.



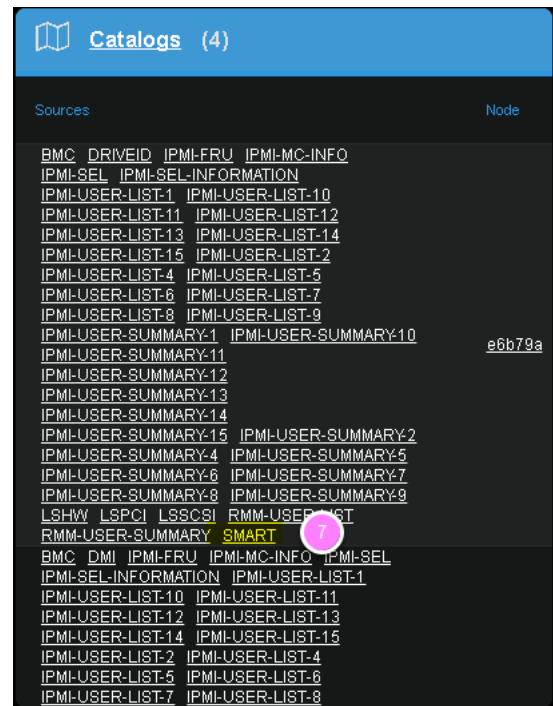
4. Click a compute node in the Node List.



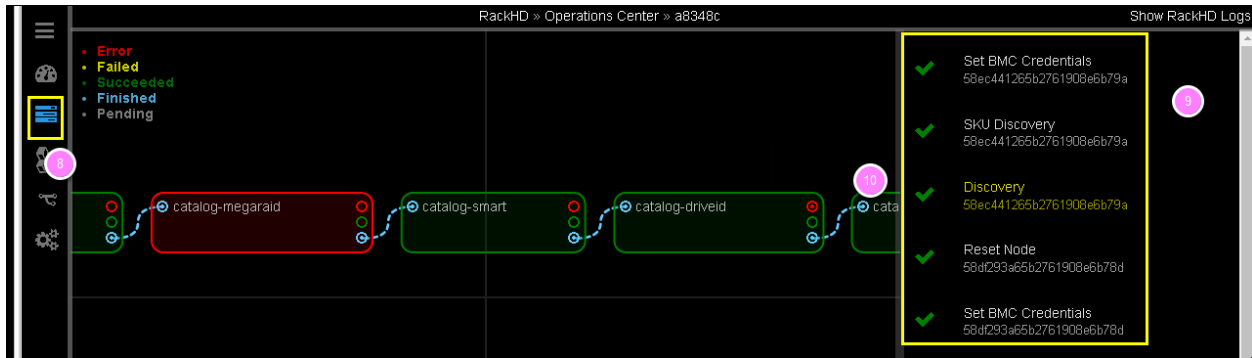
5. In the right panel, you can view the different APIs that are available, such as pollers, catalogs, and so on.
6. Experiment with the catalog of a node by clicking the “Catalogs” button.



- Try one of the catalogs link shown in the available catalogs list. Example: click “SMART” to show the Disks S.M.A.R.T information captured on the node.



- Click the “Operations Center” icon on the left panel
- You can view the workflow history and the current running workflow status.
- Click one the the workflow (example: “Discovery”) to view the workflow diagram and status.



Step 3: Create A New Workflow

In this session, you will customize a RackHD workflow to implement your own logic.

Workflow Scenario

You have a number of new bare metal servers coming online.

- Before the OS and applications are deployed to the new servers, you want to run a quick sanity check (diagnostic) on the servers.
- Due to a special demand of your application, you want to include a temperature check and CPU frequency check in the diagnostic step.

To fulfill the demand of scenario, you can use On-Web-UI to customize a new workflow named `My_Workflow`.

This example is a simple one. However, your customized workflows can be as complex as needed.

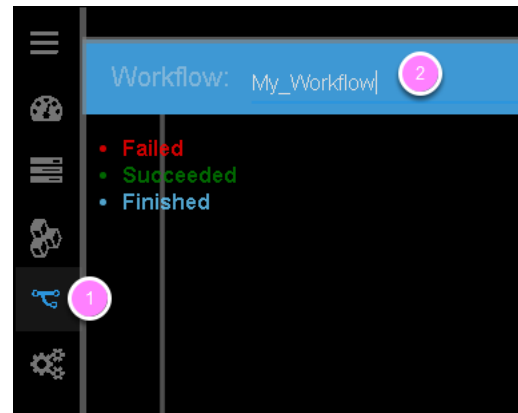
“Workflow” in RackHD

A workflow in RackHD is a JSON document, which describes a flow of execution and is built as a graph. A graph is composed by several tasks.

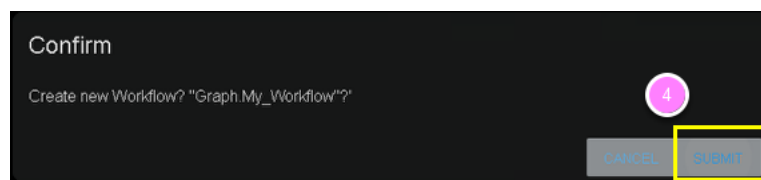
The tasks can be executed in serial or in parallel. Each task has a conditional output that can be used to drive the workflow down different paths based on how the task is completed (for example, Error, Failed, Succeeded).

Add A New Workflow

1. Click the Workflow Editor button on the left panel.
2. Type your workflow name (`My_Workflow`)
3. Press Enter on your keyboard. Do not use the Save button on the right.

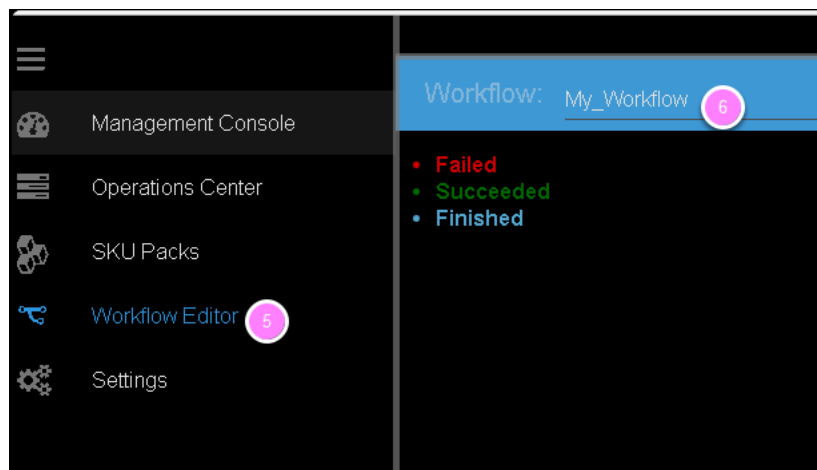


4. On the pop up Confirm dialog, click “SUBMIT”



The Web-UI refreshes itself.

5. Click the Workflow Editor button on the left panel.
6. Type My_Workflow on the name box. The name is auto-populated. You can select the workflow you created.



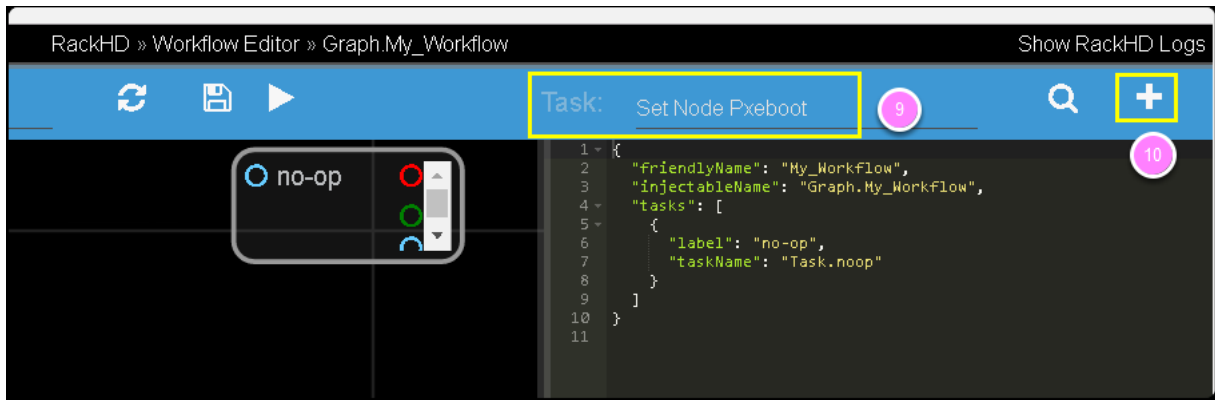
The on-web-ui will show there's a dummy operation (no-op) in this workflow.

7. Use your mouse wheel to zoom in and zoom out on the view.
8. Drag and drop from left to right to move the view point.

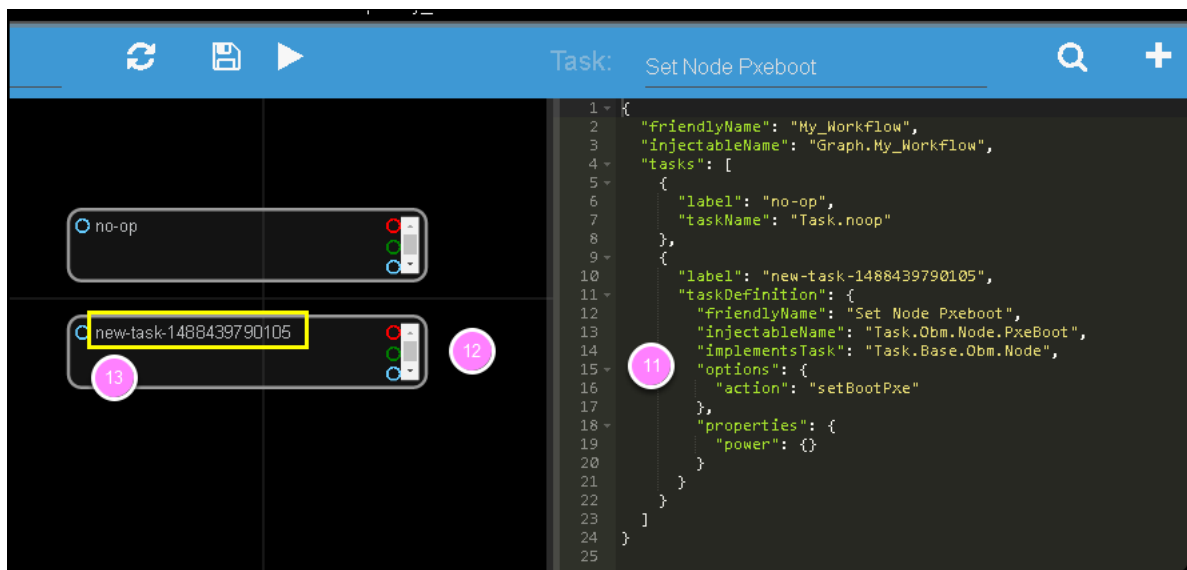


9. On the right side, above the panel that displays the workflow source code, in the Task field, type **Set Node Pxeboot**, to select an existing task.

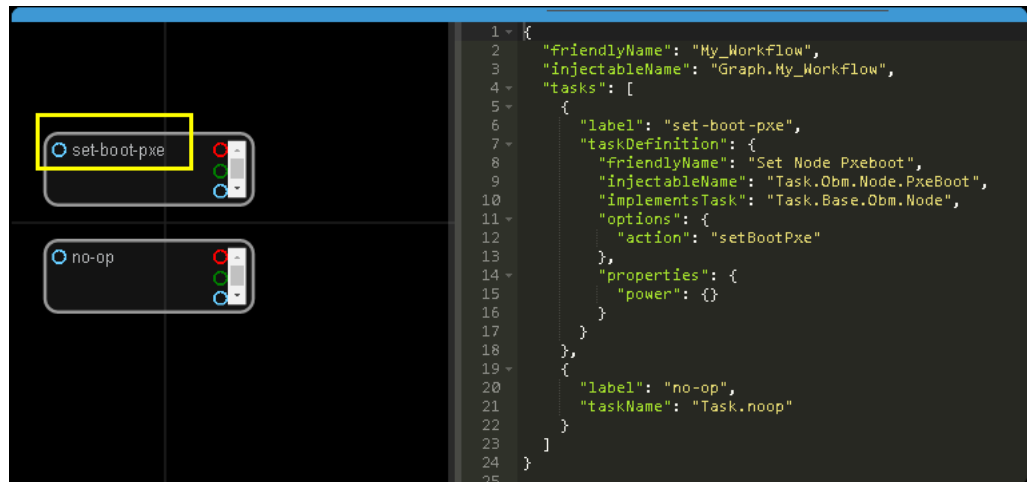
10. Click the + button, to add the task to your customized workflow.



11. Then a piece of workflow source code(json) will be appended into your workflow code .
12. On the left view, a new “task box” appeared, it will be named as “new-task-xxxxxx” (xxxxxx is randomly generated)
13. To make the name more readable, please change the label name from “new-task-xxxxxx” to “set-boot-pxe” (by clicking the string on the box then you can edit it.)



14. As below example, the newly added box has been renamed to **set-boot-pxe**.

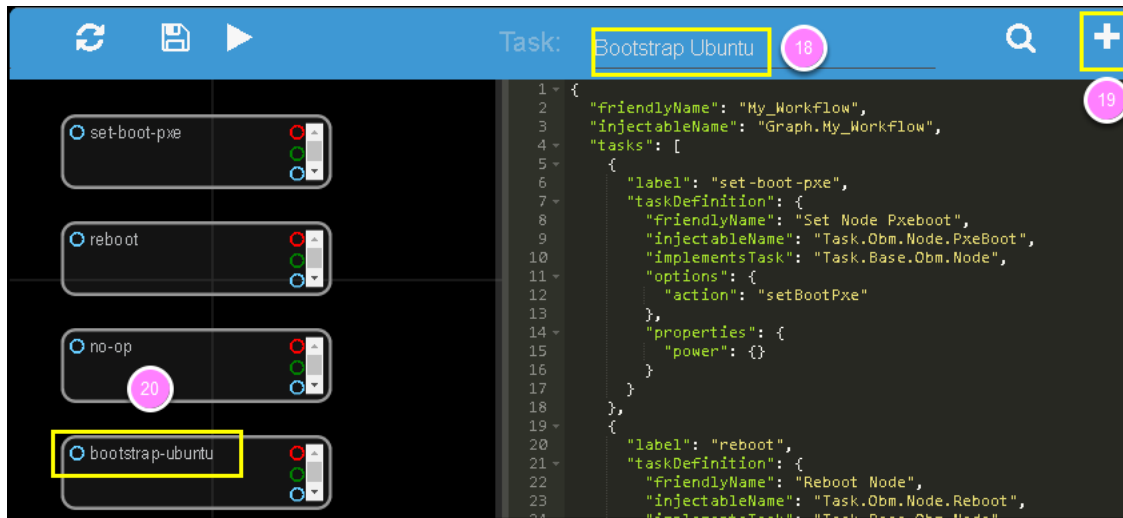


15. Select the existing task Reboot Node.
16. Click the + button. The new task is added to the source code and a new task box is added to the visual editor.
17. Change the box name from random generated string to reboot.



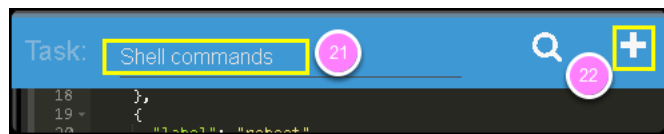
[Note] Besides, you need to edit the code block of **Reboot**, as is shown in the picture above.

18. Select the existing task Bootstrap Ubuntu
19. Click the + button.
20. Change the newly added box name to bootstrap-ubuntu

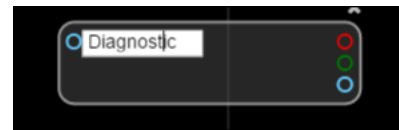


Customize A Shell Command Task

1. In the Task field, type Shell commands, to add a task.
2. Click the + button.



3. Change the new task's name to Diagnostic by clicking the name on the box.



4. In the workflow editor window on the right hand side, you can see three default shell commands for the Diagnostic task that you created.

The following example shows the default, automatically generated, json output.

```
"commands": [
  {
    "command": "sudo ls /var",
    "catalog": {
      "format": "raw",
      "source": "ls var"
    }
  },
  {
    "command": "sudo lshw -json",
    "catalog": {
      "format": "json",
      "source": "lshw user"
    }
  },
  {
    "command": "test",
```

```

    "acceptedResponseCodes": [ 1 ]
  }
]

```

```

62 - },
63 - },
64 - {
65 -   "label": "Diagnostic",
66 -   "taskDefinition": {
67 -     "friendlyName": "Shell commands",
68 -     "injectableName": "Task.Linux.Commands",
69 -     "implementsTask": "Task.Base.Linux.Commands",
70 -     "options": {
71 -       "commands": [
72 -         {
73 -           "command": "sudo ls /var",
74 -           "catalog": {
75 -             "format": "raw",
76 -             "source": "ls var"
77 -           }
78 -         },
79 -         {
80 -           "command": "sudo lshw -json",
81 -           "catalog": {
82 -             "format": "json",
83 -             "source": "lshw user"
84 -           }
85 -         },
86 -         {
87 -           "command": "test",
88 -           "acceptedResponseCodes": [
89 -             1
90 -           ]
91 -         }
92 -       ]
93 -     },
94 -     "properties": {
95 -       "commands": {}
96 -     }
97 -   }
98 - }

```

Set The Task Relationship

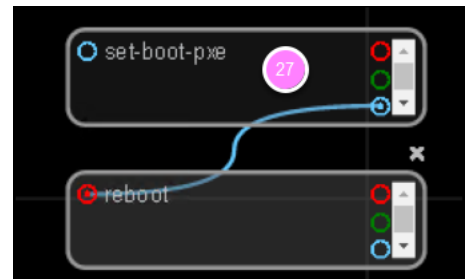
Tasks display indicators that you can connect to set the task relationship. Each task displays a trigger indicator in the top left.

Each task also displays the following condition indicators on the right side:

- Red: when fail
- Green: when success
- Blue: when finish

For example, when you connect the green condition indicator of task A to the trigger indicator for Task B: when task A has succeeded, then task B is triggered.

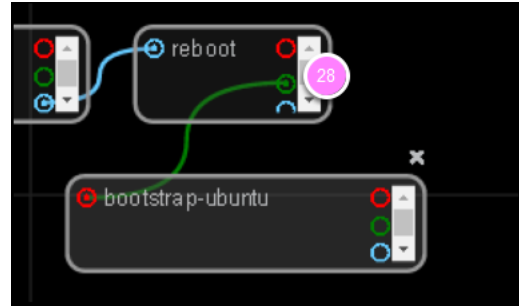
1. Connect the blue condition indicator of the set-boot-pxe task to the trigger indicator of the reboot task: whether the set-boot-pxe task is successful or not, the reboot task is triggered



2. Connect the green condition indicator of the reboot task to the trigger indicator of the bootstrap-ubuntu task.

When the reboot task is successfully completed, the bootstrap-ubuntu task is started.

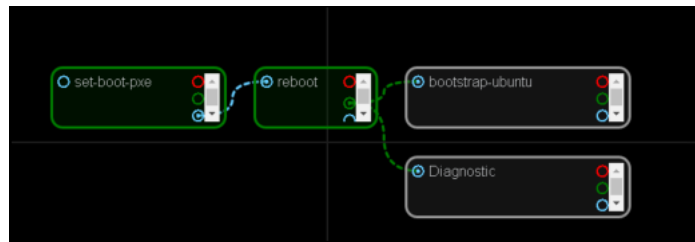
Note: Use your mouse wheel to zoom in and zoom out on the view. Drag and drop from left to right to move the view point.



3. Click x to remove the no-op task.



4. Connect the green condition indicator for the reboot task to the trigger indicator for the Diagnostic task.
5. View your new workflow.



Save The Workflow

1. Click the save icon to save the workflow



Step 4: Run The New Workflow

Click the run icon, to run the workflow that you created in 7.5.4.

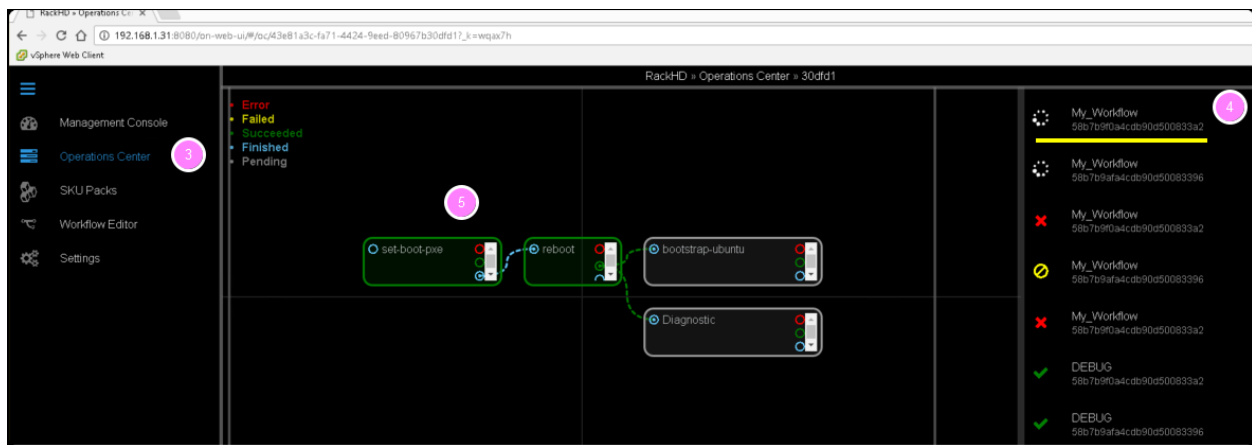


On the pop up diagram,

1. Select a node (Note: choose a compute node identified with a MAC address, instead of an Enclosure Node.)
2. Click **SAVE** to run this workflow



3. On the desktop, double-click the UltraVNC Viewer tool, to check the bootstrap progress of the node you sent this workflow to.
4. Click the Operations Center tab. You can see that "My_Workflow" is running. The target node ID is under the workflow name.
5. Click the running My_Workflow, to view the progress. After several minutes, the workflow is completed, and the color of the workflow indicates the running result (red for fail, yellow for canceled, green for success).



Perform an Unattended OS Install

Overview

In this section, you will leverage RackHD to perform an unattended install of CentOS 7 onto a discovered compute node. The config.json file for this demo already has an http proxy server set up for <http://mirror.centos.org/>. The install workflow will use this proxy to install the OS image.

Prerequisite

The compute vNode discovered in the previous section will be used as the OS-Install target node in this Lab. Get the information of vnode.

```
curl 127.0.0.1:9090/api/2.0/nodes?type=compute | jq '.'
```

```

2017-10-25 16:36:53 ① florida in ~
○ → curl 127.0.0.1:9090/api/2.0/nodes?type=compute | jq '.'
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %             0         0      10768         0 --:--:-- --:--:-- --:--:-- 10924
[
  {
    "autoDiscover": false,
    "catalogs": "/api/2.0/nodes/59f0f2debaa9750100fb2d97/catalogs",
    "id": "59f0f2debaa9750100fb2d97",
    "identifiers": [
      "52:54:be:f5:a6:2e"
    ],
    "name": "52:54:be:f5:a6:2e",
    "obms": [
      {
        "service": "ipmi-obm-service",
        "ref": "/api/2.0/obms/59f0f33abaa9750100fb2dc0"
      }
    ],
    "tags": "/api/2.0/nodes/59f0f2debaa9750100fb2d97/tags",
    "pollers": "/api/2.0/nodes/59f0f2debaa9750100fb2d97/pollers",
    "relations": [
      {
        "relationType": "enclosedBy",
        "info": null,
        "targets": [
          "59f0f32ebaa9750100fb2db3"
        ]
      }
    ],
    "sku": null,
    "type": "compute",
    "workflows": "/api/2.0/nodes/59f0f2debaa9750100fb2d97/workflows",
    "ibms": []
  }
]

```

The node-id specified in the response will be used in the following steps.

Set Up OS Mirror

To provision the OS to the node, RackHD can act as an OS mirror repository.

1. Download CentOS iso: CentOS-7-x86_64-Everything-1708.iso, from <https://www.centos.org/download/>.
2. Create OS mirror from an ISO image by typing below command. (Note: The iso file supposes to be downloaded in ~/iso)

```

cd ./RackHD/example/rackhd/files/mount/common/
mkdir -p centos/7/os/x86_64/
sudo mount -o loop ~/iso/CentOS-7-x86_64-Everything-1708.iso ./centos/7/os/x86_64/

```

Install OS with RackHD API

In this step, you will create a payload file, and then leverage the RackHD build-in workflow to install the OS on the vNode.

1. Create a payload json file by for the OS install.

Create a file named `install_centos_7_payload.json` and add the following to it:

```
{
  "name": "Graph.InstallCentOS",
  "options": {
    "defaults": {
      "version": "7",
      "repo": "http://172.31.128.2:9090/common/centos/7/os/x86_64",
      "rootPassword": "RackHDRocks!"
    }
  }
}
```

2. Install the OS by using built-in “Graph.InstallCentOS” workflow and the <node-ID> that you obtained in the Prerequisites at the beginning of this lab. Run the following command

```
curl -X POST -H 'Content-Type: application/json' -d @install_centos_7_payload.json 127.0.0.1:9090/ap
```

Installation Progress

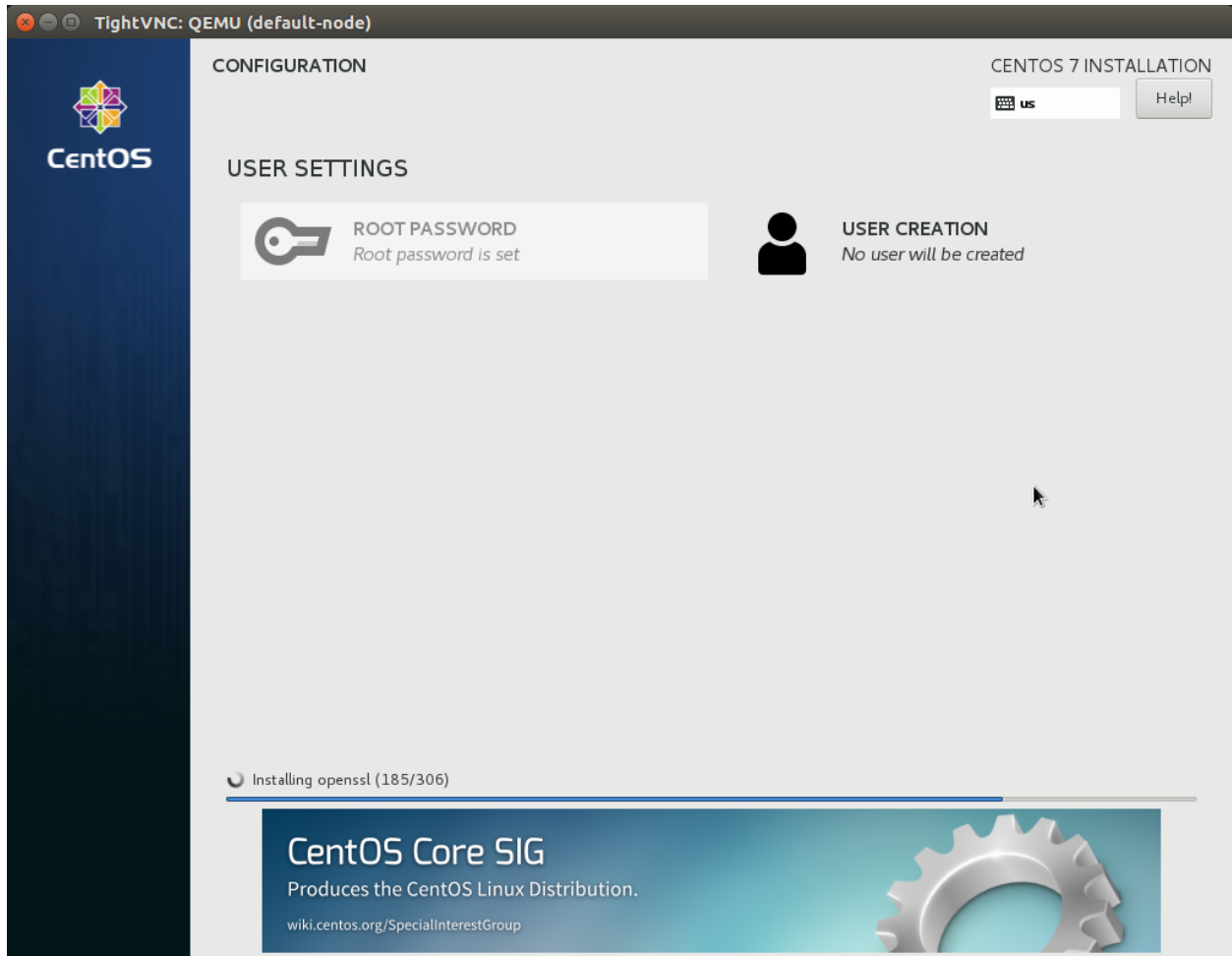
1. Run the following API to monitor a workflow that is running.

```
root@rackhd:/home/vagrant# curl 127.0.0.1:9090/api/2.0/nodes/<Node_ID>/workflows?active=true | jq '
```

In the json output RackHD responses, you will see “_status” field is “running”, and “graphName” field is “Install Cent OS”,

Note: If the “[]” is returned, the workflow failed immediately. The failure is likely because the OBM setting is not set. (No OBM service assigned to this node.)

2. Use VNC to monitor the corresponding vNode’s console.
3. It will PXE boot from the CentOS install image. And start the OS installation:



4. After the OS install has completed, you can now log into the system using the rootPassword you specified in the payload file above.

Conclusion

Thank you for taking the time to learn about RackHD.

This lab provided of brief hands-on experience of RackHD and highlighted several typical use cases. RackHD has a lot more functionality that could not be presented in this lab for the sake of time. In addition to what is currently available, there is an active community of developers who are working to continuously extend this functionality.

We encourage you to take the time to explore the links below and reach out via Slack to learn more about RackHD.

- General portal : <http://rackhd.io/>
- Documents : <http://rackhd.readthedocs.io/en/latest/index.html>
- Code in GitHub : <https://github.com/RackHD>

RackHD Workflow Engine

This tutorial explains how to use RackHD on-taskgraph as a stand alone service, known as the RackHD Workflow Engine.

Prerequisites

The Workflow Engine requires mongodb, rabbitmq, and ipmitool to be installed as follows

```
sudo apt-get install mongodb
sudo apt-get install rabbitmq-server
sudo apt-get install ipmitool
```

Set up Workflow Engine Service

1. Clone the on-taskgraph repository

```
git clone https://github.com/RackHD/on-taskgraph
cd on-taskgraph
```

2. Install the Workflow Engine dependencies

```
npm install
```

3. Copy the Workflow Engine `config.json` file to the `/opt/monorail` directory.
4. Edit the `config.json` file, and change the value of the `taskGraphEndpoint` address to the IP address of your system.
5. Start the Workflow Engine Service by using this command:

```
node index.js
```

6. Display the complete Workflow Engine API by pasting the following URL into a web browser:

```
http://<your IP address>:9030/docs/
```

7. The Workflow Engine requires DHCP, TFTP, and static file servers. You can install the RackHD [on-dhcp-proxy](#), [on-tftp](#), and [on-http](#) services respectively, as explained in their associated README files.
8. Alternatively, you can use third party versions of DHCP and TFTP as described in [TFTP and DHCP Service Setup](#)
9. You can also setup a third party static file server as described in [Static File Service Setup](#)
10. Configure your compute node to PXE boot, and reboot the node. The Workflow Engine should discover the node and catalog it in its database.

Posting a OS Install Workflow

You will need to get the discovered node's identifier from the Workflow Engine's database as follows:

```
mongo pxe
db.nodes.find().pretty()
ctrl-d
```

The output will look like:

```
{
  "name" : "52:54:be:ef:c6:85",
  "identifiers" : [
    "52:54:be:ef:c6:85"
  ],
  "type" : "compute",
  "autoDiscover" : false,
  "relations" : [ ],
```

```
"tags" : [ ],
"createdAt" : ISODate("2017-11-06T21:42:11.406Z"),
"updatedAt" : ISODate("2017-11-06T21:42:11.406Z"),
"_id" : ObjectId("5a00d7336eb470a806c2b341")
}
```

In this example, the node identifier is 5a00d7336eb470a806c2b341

Use the following command to run an OS installation workflow install using Workflow Engine,

```
curl -X POST -d @payload.json http://<ip>:<port>/api/2.0/workflows --header "Content-Type: appli
```

where, payload.json is located in the current directory level, and payload looks like the example below.

```
{
  "name": "Graph.InstallCoreOS",
  "options": {
    "defaults": {
      "graphOptions": {
        "target": "5a00d7336eb470a806c2b341"
      }
      "version": "899.17.0",
      "repo": "http://172.31.128.1:9030/coreos"
    }
  }
}
```

RackHD is a Trademark of Dell EMC Corporation.