Docker Resource Manager

Version: 0.2

Date: 6/11/2017

Contents

History

Date	Description	Editor
29/10/2017	Initial Draft	Brian Naughton

1 Introduction

The Docker resource manager is a reference implementation of the Open Service Lifecycle Manager resource manager API and specification [insert reference]. A resource manager is responsible for managing a catalog of resource packages that can be instantiated on virtual or physical infrastructure locations, e.g. OpenStack, VMWare, Kubernetes regions or locations.

This resource manager is a reference implementation intended for demonstration and educational use cases rather than production deployment. The main requirements are to

- Run a fully functional resource manager environment on a single laptop/server environment
- Provide a resource manager implementation worked example
- Sample resource packages and onboarding process

Docker is used as the virtual infrastructure manager since it can be deployed to a relatively small single host environment and provide networking and compute management. The resource manager reference implementation integrates to the Docker server APIs and presents a compliant Open Service Lifecycle Manager interface to external systems.

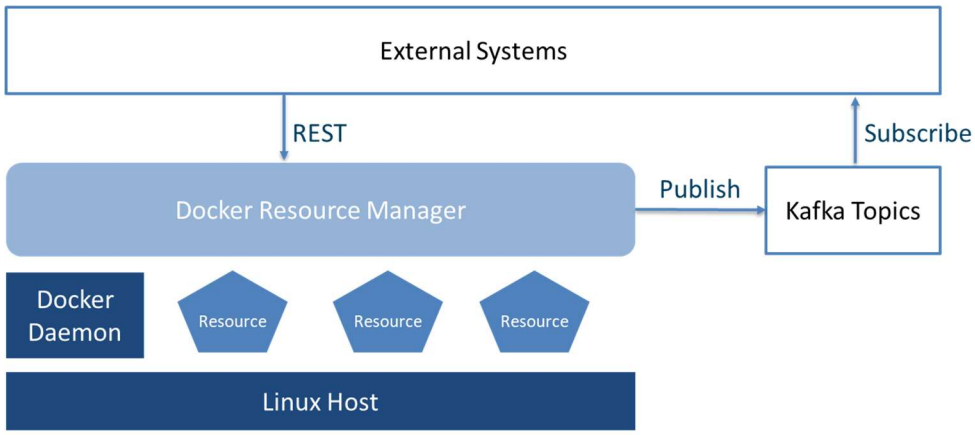

Figure 1 Docker Resource Manager

The resource manager allows external systems to create and interact with resource instances and their lifecycle transitions through a set of open interfaces.

Resource packages are bundles of application software delivering the functionality and lifecycle transition implementation required to allow external systems to fully manage a resource lifecycle. Resource packages are typically specific to the resource manager and target infrastructure it will be deployed to.

2 Actors and use cases

The following actors and use cases provide the main set of requirements on the resource manager reference implementation.

The actors are outlined below

- ***Resource developer:*** responsible for packaging network applications with their lifecycle software in a format that can be onboarded into the resource manager.
- ***Resource Manager developer:*** integration or adaptation of existing general or specific resource manager to comply with resource manager interface
- ***External systems:*** systems that instruct resource transitions and systems that monitor performance and health of resources and infrastructure

The following sections walk through the main use cases for each of the actors.

2.1 Resource developer use cases

- Wrap software in lifecycle and docker package
- Onboard resource package to resource manager
- Test resource package is working

2.2 Resource manager developer use cases

- Learn how to wrap or integrate to own system
- Test resource manager is compliant with open service lifecycle specs

2.3 External system use cases

- Discover resources in RM
- Manage resource instances through lifecycle API
- Listen for resource manager events
- Listen for resource metrics

3 Resource manager overview

This section describes the docker resource manager functional architecture and its interaction points with the actors set out in the previous section and the Docker based virtual environment it manages.

3.1 Functional Architecture

The resource manager and virtualised infrastructure is intended to run on a single laptop or server environment. The figure below shows the functional architecture for this environment.

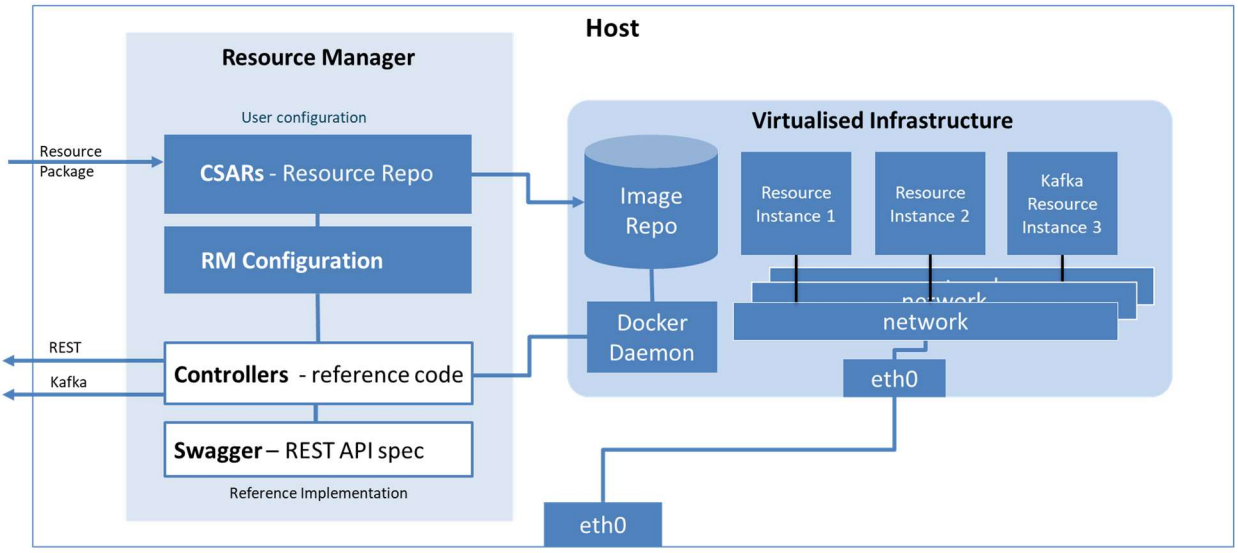

Figure 2 High level functional architecture

A Docker environment is installed as per its instructions on the target host. If the target host is linux based, then docker is installed natively on the host. If the target host is not linux based, then the docker environment should be run in a virtual machine. See further details later [reference]

The docker resource manager software can be run natively from the command line on the host or itself run within a docker container and configured appropriately to connect to the docker environment.

3.1.1 Resource manager functions

As show in figure 2, the resource manager functions can be separated into two logical groups; first the user configurable components that allow the user to onboard resources and configure the resource manager, and second the reference implementation software that implements the open service lifecycle management specifications.

The user configurable function is broken down furthers as follows:

- ***CSAR Resource Repo:*** a repository of user created resource packages that are onboarded to the docker resource manager. Each resource package contains a description of how the resource should be managed and all its software and docker build scripts.
- ***RM Configuration:*** a set of configuration files that allow the user to configure the resource manager environment, locations and system names

The reference implementation function can be broken down further as follows:

- ***Swagger API:*** the resource manager API every implementation must comply with can be found on the opensource service lifecycle management project at [reference]
- ***Controllers:*** the resource manager reference implementation code that implements the swagger API and integrates directly to the docker virtualised environment

3.1.2 Docker virtualised infrastructure

The following docker virtual infrastructure components are utilised by the resource manager:

- ***Image Repo:*** resources in the CSAR resource repository are built into docker images located in the local docker image repository
- ***Docker daemon:*** resource manager communicates with the docker daemon to manage docker images and networks
- ***Docker networks:*** docker networks are themselves exposed as resources that can be created and deletes. Docker networks can be attached to onboarded docker resource instances
- ***Resource Instances:*** images in the local docker repository can be started by the docker daemon and then further managed by the resource manager until it is removed.

3.2 Anatomy of a Docker Resource

Every resource deployed to the docker environment must be able to communicate with the resource manager to run lifecycle or operational commands, receive and update properties, and publish performance and integrity metrics.

A Docker resource framework is provided to facilitate resource manager communication and to provide a set of utilities to help resource developers package their software.

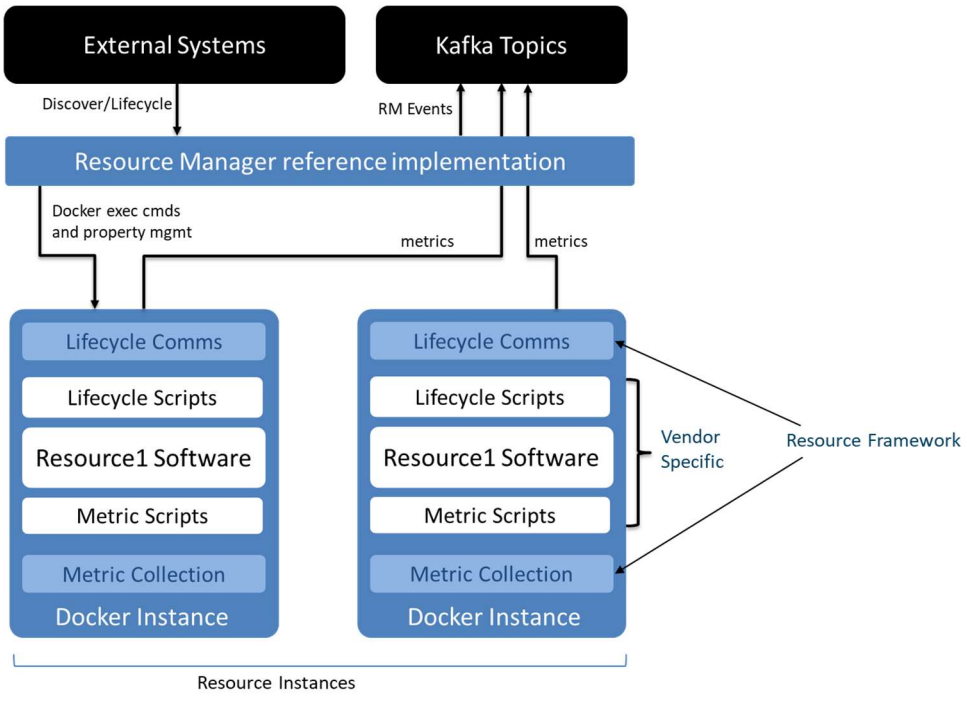

Figure 3 Docker Resource framework

As shown in the figure above, there are two main capabilities provided by the resource framework

- ***Lifecycle Comms:*** every lifecycle transition request to the resource manager results in a set of properties being passed to the docker resource followed by a docker exec command to run the appropriate transition software. A set of software utilities are provided to allow resource developers to get and set properties and interact with the resource manager.
- ***Metric Collection:*** each resource instance runs a background process that can be easily configured to capture and publish system or resource specific metrics to the resource manager or Kafka topics.

The resource framework is provided as a base docker image, allowing resource developers to extend it with their resource specific content.

Resource specific content can be broken into the following items:

- ***Resource software:*** the software to be managed
- ***Lifecycle Scripts:*** the software or scripts that implement the set of lifecycle transitions as defined by the opensource service lifecycle management specifications.
- ***Metric Scripts:*** software or scripts to monitor the resource software or docker container resources

3.2.1 Base Docker resource image

The software that makes up the base docker resource image can be found at **<RM_INSTALL_DIR>/opt/csars/baseimage**. A docker-compose file is provided to build the base image at **<RM_INSTALL_DIR>/rm-base.yml**.

Running the following docker compose command will build a docker image called **dockerrm_baseimage** in the local docker image repository.

```
docker-compose -f rm-base.yml build
```

This image can be extended by resource developers as the basis for their resource packages, see [reference].

3.2.1.1 Base Docker file

The base image docker file is shown below.

```
FROM ubuntu:latest
# base software dependencies
RUN apt-get update && apt-get install -y …
RUN pip3 install Jinja2 urllib3 netifaces netaddr
```

The base image is based on latest ubuntu docker image and firstly loads all dependent software packages to support the base docker image utility commands described in the next sections.

```
# copy utilities to base image
COPY utilities/* /usr/local/bin/
RUN chmod -R +x /usr/local/bin/*
```

The utility commands themselves are copied

```
# copy collectd configuration
COPY collectd.conf /etc/collectd/
```

Collectd configuration is copied in preparation for running collectd in the entrypoint script. This will be discussed in the next sections.

```
# add entrypoint
ADD entrypoint.sh /entrypoint.sh
```

```
RUN chmod +x /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
```

3.2.1.2 Lifecycle Comms

Every transition called by the resource manager results in a sequence of docker exec commands being issued to the appropriate docker resource instance. An initial docker exec command pushes the property values associated with the transition from the resource manager to a yaml file at **/opt/rmparams** inside the resource instance container. Once this is complete a further docker exec command is run that calls the specified transition method itself. In this way making the properties available to the lifecycle scripts of software to be dealt with as needed.

The resource manager sends a final docker exec command to re-read the params, updating its properties if needed.

Properties from lifecycle and operation requests from the resource manager to the resource instance are handled differently to avoid conditions where either sets of properties are overwritten by each other. Operation parameters are stored inside the resource container at **/opt/opparams.**

Utility commands are available in the base image to allow the resource instance to easily access and update the last lifecycle or operation properties sent from the resource manager. The following utility commands are available.

Name	Description
getProperty	Get the value of named lifecycle property
setProperty	Set the value of named lifecycle property
getOperationProperty	Get the value of named operation property
setOperationProperty	Set the value of named operation property

The example below shows a **start** lifecycle transition which calls a bash script. The script reads a number properties it requires to run the resource software.

```
#!/bin/bash
static=`getProperty staticReference`
duration=`getProperty callDuration`
maxCalls=`getProperty maxCalls`
targetIP=`getProperty targetip`

/sipp-3.5.1/sipp -bg -sf /myuac.xml -trace_stat -fd 3 -d $duration -s 1002
$targetIP -l $maxCalls -mp 5606 > /etc/sipp.txt
```

3.2.1.3 Metric Collection

Each resource instance must run a collectd process throughout its life. This is specified in the base image entrypoint script. This can be seen below.

```
#!/bin/bash
set -e
collectd -f
```

Running collectd in this fashion has two purposes:

- Keeps the docker container alive throughout the lifecycle of the resource software
- Provide a framework for collecting performance and integrity metrics for the container infrastructure and the resource software itself.

```
def configer(ObjConfiguration):
  collectd.info('Configuring')

def initer():
  collectd.info('initing')

def read_callback(data=None):
  collectd.info('reading value')

  processname = 'asterisk'
  tmp = os.popen("ps -Af").read()
  proccount = tmp.count(processname)

  metric = collectd.Values()
  metric.plugin = 'asterisk process watcher'
  metric.host = 'asterisk'
  metric.interval = 5
  metric.type = 'gauge'
  metric.values = [proccount]
  metric.dispatch()

collectd.register_config(configer)
collectd.register_init(initer)
collectd.register_read(read_callback)
```

3.2.1.4 Network Interface Utilities

The following additional command utilities are available in the base image as helpers in writing resource instance transition scripts.

Name	Description
getInterfaces	List the network interfaces available to the resource instance docker container
getInterfacesSub	Return the subnet address of a given network address
getInterfaceIP	Return the ip address of the given network address

3.3 Built in resources

The base image provides the framework for user defined resource development. In addition to these user defined resource packages the docker resource manager provides some “built-in” resource types whose lifecycles can be manipulated in the same fashion as user provided resources.

- Docker networks:
- Docker volumes:

3.3.1 Docker networks

A built in docker-network resource type is provided which allows docker network resources to be managed with no additional development. The built-in docker network resource descriptor is shown below, this can be discovered and manipulated through the REST APIs.

```
name: resource::docker-network::1.0
description: resource package for user defined docker network
properties:
  networkname:
    type: string
    description: name of the network
    value: ${instance.name}
    required: true
  subnet:
    type: string
    description: subnet for new network
    default: "10.1.1.0/24"
  gateway:
    type: string
    description: subnet of user defined network
    default: "10.1.1.1"
  bridgename:
    type: string
    description: name of the assigned linux bridge
    read-only: true
  networkid:
    type: string
    description: docker id of network instance
    read-only: true
lifecycle:
- Install
- Uninstall
```

It's worth noting above the lifecycle transitions supported are restricted to *Install* and *Uninstall*. This method can be useful when proxying physical resources that do not support the full set of lifecycle transitions.

Existing docker networks are discovered when the resource manager starts up and can be discovered as resource instances through the resource manager rest APIs. Existing docker network lifecycles cannot be manipulated through the API

Additional docker networks can be created as normal through the resource manager REST API. User created docker networks also set the linux bridge name of the network to a human readable string to allow non docker software to bind to the network naturally, e.g. PCAP network probes can use the bridge name to bind to the Linux bridge created.

3.3.2 Docker volumes

<volumes not yet implemented>

3.4 Built in Operations

The resource manager also provides built in operations every resource instance can utilise with no need for any additional development effort. The following operations can be added to any resource instance.

Name	Description
addNetwork	Add named network to resource instance
removeNetwork	Remove named network from resource instance

For resource developers who want to allow networks to be added after the install transition the following configuration must be added to your **resource.yml** descriptor.

```
operations:
  addNetwork:
    properties:
      networkid:
        type: string
        description: built in addnetwork operation
        required: true
  removeNetwork:
    properties:
      networkid:
        type: string
        description: built in remove network operation
        required: true
```

4 Developing Resources

The figure below shows the tasks required by a resource developer to onboard a resource package into the docker resource manager.

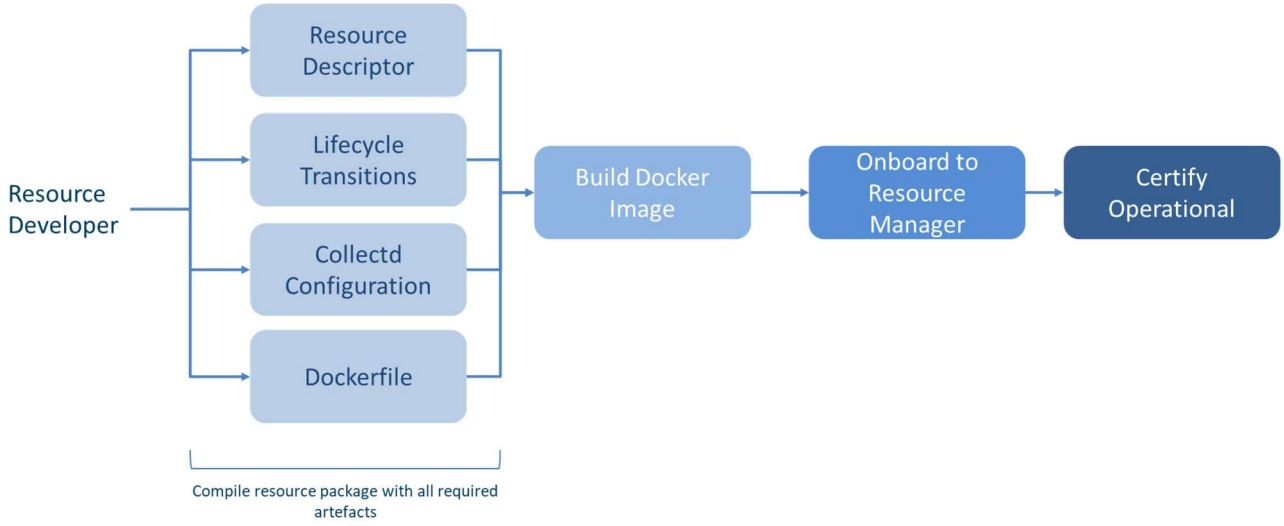

Figure 4 Docker resource onboarding process

Resource developers must create all required resource package artefacts as follows

- ***Resource descriptor:*** specifying the expected property values, supported lifecycle transitions/operations and any metrics published by the resource
- ***Lifecycle transitions:*** the software that implements the standard lifecycle transitions and/or operations
- ***Collectd configuration:*** any resource specific metric collection is specified through collectd configuration
- ***Dockerfile:*** a dockerfile that extends the base image and adds the resource software, lifecycle software, and collectd configuration files.

Once all resource artefacts are in place, the docker resource image must be built into the local docker repository and onboarded to the resource manager.

4.1 Resource package structure

Each resource package has its own directory within a top-level directory specified to the docker resource manager. Within each resource package directory, the following file structure is expected.

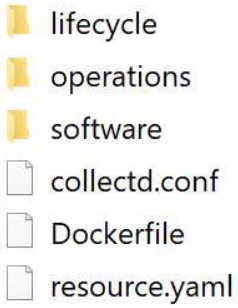

Figure 5 Resource Package Structure

4.2 Resource Descriptor

Each resource package must provide a valid resource descriptor that describes the expected lifecycle behaviour and properties required to instantiate a resource instance. Each resource descriptor can have 3 sections as follows:

- ***Properties:*** specifies a set of properties expected from external systems to support resource instance lifecycle
- ***Supported Lifecycle:*** lists the set of standard transition supported by this resource.
- ***Metrics & Policies:*** policies can be defined that reference metrics produced by resources for auto scaling and healing scenarios
- ***Operations:*** non-standard lifecycle operations and their properties are listed

As well as the above sections, each resource has basic name and description properties

```
name: resource::asterisk::1.0
description: asterisk server
```

Resource names must comply with the naming convention of <type>::<resource name>::<version>.

4.2.1 Properties

Properties are specified as described in the Resource Manager specification document [insert ref].

```
properties:
  docker_network:
    type: string
    description: name of network to attach container to on startup
    default: "life_default"
    required: true
  docker_ipaddr:
    type: string
    description: assigned ip address to this container
    read-only: true
  proxyAddress:
    type: string
    description: address of a proxy server, if empty asterisk is standalone
    default: ""
```

In addition to the standard resource property definitions, the docker resource manager imposes some additional property semantics.

Each resource manager must provide at least one ***docker_network*** property with the name of an existing docker network created by the resource manager. The current version of the resource manager expects at least one ***docker_network*** to bind the docker container to on startup.

Each ***docker_network*** property is accompanied with a read only ***docker_ipaddr*** property and during the installation transition for the resource, the docker resource manager populates the ***docker_ipaddr*** property with the IP address allocated to the resource instances docker container.

Additional docker networks can be added to the container to be setup during the installation transition by adding more ***docker_network*** and ***docker_ipaddr*** property pairs and appending a number after each. For example

```
properties:
```

```
  docker_network:
    type: string
    description: name of network to attach container on startup
    default: "life_default"
    required: true
  docker_ipaddr:
    type: string
    description: assigned ip address
    read-only: true
  docker_network2:
    type: string
    description: name of network to attach container on startup
    required: true
  docker_ipaddr2:
    type: string
    description: assigned ip address for 2nd network
    read-only: true
```

4.2.2 Standard lifecycle & Operations

This section lists the supported standard lifecycle transitions a resource provides, i.e. one or more from the following: Install, Configure, Start, Integrity, Stop, Uninstall.

For example, a resource may not provide a full set of transitions, for example the docker network resource only provides the following:

```
lifecycle:
- Install
- Uninstall
```

To map listed lifecycle transitions to software commands that execute them, there is a configuration file provided in the lifecycle directory, ***<RESOURCE_PACKAGE_DIR>/lifecycle/lifecycle.yaml***, that maps each transition to an executable software location.

```
---
version: 1
lifecycle:
  install: /opt/lifecycle/install.sh
  configure: /opt/lifecycle/configure.sh
  start: /opt/lifecycle/start.sh
  integrity: /opt/lifecycle/integrity.sh
  stop: /opt/lifecycle/stop.sh
  reconfigure: /opt/lifecycle/reconfigure.sh
  uninstall: /opt/lifecycle/uninstall.sh
```

In addition to the standard lifecycle transitions, ad-hoc operations are listed with their properties.

```
operations:
  addAsterisk:
    properties:
      ipaddr:
        type: string
        description: address of the asterisk to add to pool
        required: true
```

```
      port:
        type: string
        description: port of the asterisk to add to pool
        default: 5060
        required: true
  removeAsterisk:
    properties:
      ipaddr:
        type: string
        description: address of the asterisk to remove from the pool
        required: true
      port:
        type: string
        description: address of the asterisk to remove from the pool
        default: 5060
        required: true
```

To map the operations to software commands, there is a configuration file provided in the operations directory, ***<RESOURCE_PACKAGE_DIR>/operations/operations.yaml***

```
---
version: 1
operations:
  addAsterisk: /opt/lifecycle/addAsterisk.py
  removeAsterisk: /opt/lifecycle/removeAsterisk.py
```

4.2.3 Performance and integrity metrics

As defined in the resource manager specification [insert reference], healing and scaling metrics and policies can also be provided in the resource specification as below.

```
metrics:
  h_integrity:
    type: "metric::integrity"
    publication-period: "5"
  load:
    type: "metric::load"
policies:
  heal:
    metric: "h_integrity"
    type: "policy::heal"
    properties:
      smoothing:
        value: 3
```

4.3 Building docker images

The resource package ***Dockerfile*** must extend the base docker image and put its various software components into a set of pre-defined locations.

```
FROM dockerrm_baseimage

# copy lifecycle scripts to /opt/lifecycle
RUN mkdir /opt/lifecycle && chmod 777 /opt/lifecycle
```

```
ADD lifecycle /opt/lifecycle

# copy operation scripts to /opt/operations
ADD operations /opt/lifecycle
RUN chmod 777 /opt/lifecycle/*

# copy metric configuration
COPY collectd.conf /etc/collectd/

# copy resource software
<COPY your software here>
```

As shown above, the resource Dockerfile must extend the dockerrm_baseimage. Lifecycle and operations directories are copied to ***/opt/lifecycle***. Collectd assets must be configured through a collectd.conf file that is placed in /etc/collectd.

All custom software assets can be placed as appropriate and referenced by the lifecycle and operations scripts to be managed.

```
cd <RESOURCE_DIR>
docker build -t dockerrm_<RESOURCENAME> .
```

Once the ***Dockerfile*** is in place, the resource docker image must be built with a ***dockerrm_*** prefix to allow external systems to differentiate the images from others.

5 Resource Manager Administration

This section details administrative configuration options for the docker resource manager.

5.1 System Configuration

The locations supported by this instance of the resource manager are configured in **<RESOURCE_MANAGER_ DIR>/opt/rm/config/locations.yml** file.. Multiple locations can be specified.

```
---
locations:
  - name: dev
    type: development
    description: "development cloud"
```

System wide configuration options can set in the **<RESOURCE_MANAGER_ DIR>/opt/rm/config/config.yml** file.

```
---
name: "docker-rm::dev"
version: "1.0.0"
supportedAPIVersions:
  - "1.0"
```

Name and version of the API supported can be specified.

```
csardirs:
  - "csars"
```

A list of CSAR directories are provided to tell the docker resource manager where to look for resource packages.

```
supportedFeatures:
  asynchronousTransitionResponse: True
properties:
  responseKafkaConnectionUrl: "kafka:9092"
  responseKafkaTopicName: "docker-rm"
```

To enable kafka responses to tasks this section must be configured appropriately with the correct kafka location and topic name.

```
  generateTraceFile: False
  traceFile: /opt/rm/logs/trace.csv
```

A trace file can also be enabled that will dump all request and response messages to/from the docker resource manager to a CSV file at the above location.

5.2 Logging

Logging levels can set in the **<RESOURCE_MANAGER_ DIR>/opt/rm/config/logging.yml** file. By default logging levels are set to INFO.

```
loggers:
    controllers:
        level: INFO
        handlers: [info_file_handler, error_file_handler]
        propagate: no
```