Apache Karaf Decanter 1.x - Documentation

Apache Software Foundation

Apache Karaf Decanter 1.x - Documentation

User Guide

1. Introduction

Apache Karaf Decanter is monitoring solution running in Apache Karaf.

It’s composed in three parts:

- Collectors are responsible of harvesting monitoring data. Decanter provides collectors to harvest different kind of data. We have two kinds of collectors:
    - Event Driven Collectors automatically react to events and send the event data to the Decanter appenders.
    - Polled Collectors are periodically called by the Decanter Scheduler. They harvest data and send it to the Decanter appenders
- Appenders receive the data from the collectors and are responsible to store the data into a given backend. Decanter provides appenders depending of the backend storage that you want to use.
- SLA is a special kind of appender. It receives all harvested data and checks on it. If a check fails, an alert event is created and sent to alerters. Decanter provides alerters depending of the kind of notification that you want.

Apache Karaf Decanter provides Karaf features for each collector, appender, SLA alerter.

The first thing to do is to add the Decanter features repository in Karaf:

```
karaf@root()> feature:repo-add mvn:org.apache.karaf.decanter/apache-karaf-decanter/1.0.0/
xml/features
```

Now, you have to install the collectors, appenders, and eventually SLA alerters feature to match your need.

2. Collectors

Decanter collectors harvest the monitoring data, and send this data to the Decanter appenders.

Two kinds of collector are available:

- Event Driven Collectors react to events and "broadcast" the data to the appenders.
- Polled Collectors are periodically executed by the Decanter Scheduler. When executed, the collectors harvest the data and send to the appenders.

2.1. Log

The Decanter Log Collector is an event driven collector. It automatically reacts when a log occurs, and send the log details (level, logger name, message, etc) to the appenders.

The `decanter-collector-log` feature installs the log collector:

```
karaf@root()> feature:install decanter-collector-log
```

The log collector doesn’t need any configuration, the installation of the decanter-collector-log feature is enough.

2.2. File

The Decanter File Collector is an event driven collector. It automatically reacts when new lines are appended into a file (especially a log file). It acts like the tail Unix command. Basically, it’s an alternative to the log collector. The log collector reacts for local Karaf log messages, whereas the file collector can react to any files, included log file from other system than Karaf. It means that you can monitor and send collected data for any system (even not Java base, or whatever).

The file collector deals with file rotation, file not found.

The `decanter-collector-file` feature installs the file collector:

```
karaf@root()> feature:install decanter-collector-file
```

Now, you have to create a configuration file for each file that you want to monitor. In the etc folder, you have to create a file with the following format name `etc/org.apache.karaf.decanter.collector.file-ID.cfg` where ID is an ID of your choice.

This file contains:

```
type=my
path=/path/to/file
any=value
```

- `type` is an ID (mandatory) that allows you to easily identify the monitored file
- `path` is the location of the file that you want to monitore
- all other values (like `any`) will be part of the collected data. It means that you can add your own custom data, and easily create queries bases on this data.

For instance, instead of the log collector, you can create the following `etc/org.apache.karaf.decanter.collector.file-karaf.cfg` file collector configuration file:

```
type=karaf-log-file
path=/path/to/karaf/data/log/karaf.log
my=stuff
```

The file collector will tail on karaf.log file, and send any new line in this log file as collected data.

2.3. JMX

The Decanter JMX Collector is a polled collector, executed periodically by the Decanter Scheduler.

The JMX collector connects to a JMX MBeanServer (local or remote), and retrieves all attributes of each available MBeans. The JMX metrics (attribute values) are send to the appenders.

The `decanter-collector-jmx` feature installs the JMX collector, and a default configuration file:

```
karaf@root()> feature:install decanter-collector-jmx
```

This feature brings a `etc/org.apache.karaf.decanter.collector.jmx-local.cfg` configuration file containing:

```
#
# Decanter Local JMX collector configuration
#

# Name/type of the JMX collection
type=jmx-local

# URL of the JMX MBeanServer.
# local keyword means the local platform MBeanServer or you can specify to full JMX URL
# like service:jmx:rmi:///jndi/rmi://hostname:port/karaf-instance
url=local

# Username to connect to the JMX MBeanServer
#username=karaf

# Password to connect to the JMX MBeanServer
#password=karaf

# Object name filter to use. Instead of harvesting all MBeans, you can select only
# some MBeans matching the object name filter
#object.name=org.apache.camel:context=*,type=routes,name=*
```

This file harvests the data of the local MBeanServer:

- the `type` property is a name (of your choice) allowing you to easily identify the harvested data
- the `url` property is the MBeanServer to connect. "local" is reserved keyword to specify the local MBeanServer. Instead of "local", you can use the JMX service URL. For instance, for Karaf version 3.0.0, 3.0.1, 3.0.2, and 3.0.3, as the local MBeanServer is secured, you can specify `service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root`. You can also polled any remote MBean server (Karaf based or not) providing the service URL.
- the `username` property contains the username to connect to the MBean server. It's only required if the MBean server is secured.
- the `password` property contains the password to connect to the MBean server. It's only required if the MBean server is secured.
- the `object.name` property is optional. If this property is not specified, the collector will retrieve the attributes of all MBeans. You can filter to consider only some MBeans. This property contains the ObjectName filter to retrieve the attributes only to some MBeans.
- any other values will be part of the collected data. It means that you can add your own property if you want to add additional data, and create queries based on this data.

You can retrieve multiple MBean servers. For that, you just create a new configuration file using the file name format `etc/org.apache.karaf.decanter.collector.jmx-[ANYNAME].cfg`.

2.4. ActiveMQ (JMX)

The ActiveMQ JMX collector is just a special configuration of the JMX collector.

The `decanter-collector-activemq` feature installs the default JMX collector, with the specific ActiveMQ JMX configuration:

```
karaf@root()> feature:install decanter-collector-activemq
```

This feature installs the same collector as the `decanter-collector-jmx`, but also add the `etc/org.apache.karaf.decanter.collector.jmx-activemq.cfg` configuration file.

This file contains:

```
#
# Decanter Local ActiveMQ JMX collector configuration
#

# Name/type of the JMX collection
type=jmx-activemq

# URL of the JMX MBeanServer.
# local keyword means the local platform MBeanServer or you can specify to full JMX URL
# like service:jmx:rmi:///jndi/rmi://hostname:port/karaf-instance
url=local

# Username to connect to the JMX MBeanServer
#username=karaf

# Password to connect to the JMX MBeanServer
#password=karaf

# Object name filter to use. Instead of harvesting all MBeans, you can select only
# some MBeans matching the object name filter
object.name=org.apache.activemq:*
```

This configuration actually contains a filter to retrieve only the ActiveMQ JMX MBeans.

2.5. Camel (JMX)

The Camel JMX collector is just a special configuration of the JMX collector.

The `decanter-collector-camel` feature installs the default JMX collector, with the specific Camel JMX configuration:

```
karaf@root()> feature:install decanter-collector-camel
```

This feature installs the same collector as the `decanter-collector-jmx`, but also add the `etc/org.apache.karaf.decanter.collector.jmx-camel.cfg` configuration file.

This file contains:

```
#
# Decanter Local Camel JMX collector configuration
#

# Name/type of the JMX collection
type=jmx-camel

# URL of the JMX MBeanServer.
# local keyword means the local platform MBeanServer or you can specify to full JMX URL
# like service:jmx:rmi:///jndi/rmi://hostname:port/karaf-instance
url=local

# Username to connect to the JMX MBeanServer
#username=karaf

# Password to connect to the JMX MBeanServer
#password=karaf

# Object name filter to use. Instead of harvesting all MBeans, you can select only
# some MBeans matching the object name filter
object.name=org.apache.camel:context=*,type=routes,name=*
```

This configuration actually contains a filter to retrieve only the Camel Routes JMX MBeans.

2.6. Camel Tracer

The Camel Tracer provides a Camel Tracer Handler that you can set on a Camel Tracer.

If you enable the tracer on a Camel route, all tracer events (exchanges on each step of the route) are send to the appenders.

The `decanter-collector-camel-tracer` feature provides the Camel Tracer Handler:

```
karaf@root()> feature:install decanter-collector-camel-tracer
```

Now, you can use the Decanter Camel Tracer Handler in a tracer that you can use in routes.

For instance, the following route definition shows how to enable tracer on a route, and use the Decanter Tracer Handler in the Camel Tracer:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

    <reference id="eventAdmin" interface="org.osgi.service.event.EventAdmin"/>

    <bean id="traceHandler"
class="org.apache.karaf.decanter.collector.camel.DecanterTraceEventHandler">
        <property name="eventAdmin" ref="eventAdmin"/>
    </bean>

    <bean id="tracer" class="org.apache.camel.processor.interceptor.Tracer">
        <property name="traceHandler" ref="traceHandler"/>
        <property name="enabled" value="true"/>
        <property name="traceOutExchanges" value="true"/>
        <property name="logLevel" value="OFF"/>
    </bean>

    <camelContext trace="true" xmlns="http://camel.apache.org/schema/blueprint">
        <route id="test">
            <from uri="timer:fire?period=10000"/>
            <setBody><constant>Hello World</constant></setBody>
            <to uri="log:test"/>
        </route>
    </camelContext>

</blueprint>
```

2.7. System

The system collector is a polled collector (periodically executed by the Decanter Scheduler).

This collector executes operating system commands (or scripts) and send the execution output to the appenders.

The `decanter-collector-system` feature installs the system collector:

```
karaf@root()> feature:install decanter-collector-system
```

This feature installs a default `etc/org.apache.karaf.decanter.collector.system.cfg` configuration file containing:

```
#
# Decanter OperationSystem Collector configuration
#

# This collector executes system commands, retrieve the exec output/err
# sent to the appenders
#
# The format is key=command
# for instance:
# df=df -h
# free=free
# You can also create a script containing command like:
#
#   df -k / | awk -F " |%" '/dev/{print $8}'
#
# This script will get the available space on the / filesystem for instance.
# and call the script:
# df=/bin/script
#
# Another example of script to get the temperature:
#
#   sensors|grep temp1|awk '{print $2}'|cut -b2,3,4,5
#
```

You can add the commands that you want to execute using the format:

```
name=command
```

The collector will execute each command described in this file, and send the execution output to the appenders.

For instance, if you want to periodically send the free space available on the / filesystem, you can add:

```
df=df -k / | awk -F " |%" '/dev/{print $8}'
```

3. Appenders

Decanter appenders receive the data from the collectors, and store the data into a storage backend.

3.1. Log

The Decanter Log Appender creates a log message for each event received from the collectors.

The `decanter-appender-log` feature installs the log appender:

```
karaf@root()> feature:install decanter-appender-log
```

The log appender doesn't require any configuration.

3.2. Elasticsearch

The Decanter Elasticsearch Appender stores the data (coming from the collectors) into an Elasticsearch instance.

This appender transforms the data as a json object. This json object is stored in the Elasticsearch instance.

It's probably one of the most interesting way to use Decanter.

The `decanter-appender-elasticsearch` feature installs the elasticsearch appender:

```
karaf@root()> feature:install decanter-appender-elasticsearch
```

This feature installs the elasticsearch appender, especially the `etc/org.apache.karaf.decanter.appender.elasticsearch.cfg` configuration file containing:

```
################################################
# Decanter Elasticsearch Appender Configuration
################################################

# Hostname of the elasticsearch instance
host=localhost
# Port number of the elasticsearch instance
port=9300
# Name of the elasticsearch cluster
clusterName=elasticsearch
```

This file contains the elasticsearch instance connection properties:

- the `host` property contains the hostname (or IP address) of the Elasticsearch instance
- the `port` property contains the port number of the Elasticsearch instance
- the `clusterName` property contains the name of the Elasticsearch cluster where to send the data

3.2.1. Embedding Decanter Elasticsearch

For convenience, Decanter provides an `elasticsearch` feature starting an embedded Elasticsearch instance:

```
karaf@root()> feature:install elasticsearch
```

Thanks to this elasticsearch instance, by default, the decanter-appender-elasticsearch will send the data to this instance.

The feature also installs the `etc/elasticsearch.yml` configuration file:

```
################################################################################
#################### Elasticsearch Decanter Configuration ###################
################################################################################

# WARNING: change in this configuration file requires a refresh or restart of
# the elasticsearch bundle

################################### Cluster ###################################

# Cluster name identifies your cluster for auto-discovery. If you're running
# multiple clusters on the same network, make sure you're using unique names.
#
cluster.name: elasticsearch
cluster.routing.schedule: 50ms


#################################### Node #####################################

# Node names are generated dynamically on startup, so you're relieved
# from configuring them manually. You can tie this node to a specific name:
#
node.name: decanter

# Every node can be configured to allow or deny being eligible as the master,
# and to allow or deny to store the data.
#
# Allow this node to be eligible as a master node (enabled by default):
#
#node.master: true
#
# Allow this node to store data (enabled by default):
#
node.data: true

# You can exploit these settings to design advanced cluster topologies.
#
# 1. You want this node to never become a master node, only to hold data.
#    This will be the "workhorse" of your cluster.
#
#node.master: false
#node.data: true
#
# 2. You want this node to only serve as a master: to not store any data and
#    to have free resources. This will be the "coordinator" of your cluster.
#
#node.master: true
#node.data: false
#
# 3. You want this node to be neither master nor data node, but
#    to act as a "search load balancer" (fetching data from nodes,
#    aggregating results, etc.)
#
#node.master: false
#node.data: false

# Use the Cluster Health API [http://localhost:9200/_cluster/health], the
# Node Info API [http://localhost:9200/_nodes] or GUI tools
# such as <http://www.elasticsearch.org/overview/marvel/>,
# <http://github.com/karmi/elasticsearch-paramedic>,
# <http://github.com/lukas-vlcek/bigdesk> and
# <http://mobz.github.com/elasticsearch-head> to inspect the cluster state.

# A node can have generic attributes associated with it, which can later be used
# for customized shard allocation filtering, or allocation awareness. An attribute
# is a simple key value pair, similar to node.key: value, here is an example:
#
#node.rack: rack314

# By default, multiple nodes are allowed to start from the same installation location
# to disable it, set the following:
#node.max_local_storage_nodes: 1


#################################### Index ####################################

# You can set a number of options (such as shard/replica options, mapping
# or analyzer definitions, translog settings, ...) for indices globally,
# in this file.
#
# Note, that it makes more sense to configure index settings specifically for
# a certain index, either when creating it or by using the index templates API.
#
# See <http://elasticsearch.org/guide/en/elasticsearch/reference/current/
index-modules.html> and
# <http://elasticsearch.org/guide/en/elasticsearch/reference/current/
indices-create-index.html>
# for more information.

# Set the number of shards (splits) of an index (5 by default):
#
#index.number_of_shards: 5

# Set the number of replicas (additional copies) of an index (1 by default):
#
#index.number_of_replicas: 1

# Note, that for development on a local machine, with small indices, it usually
# makes sense to "disable" the distributed features:
```

```
#
#index.number_of_shards: 1
#index.number_of_replicas: 0

# These settings directly affect the performance of index and search operations
# in your cluster. Assuming you have enough machines to hold shards and
# replicas, the rule of thumb is:
#
# 1. Having more *shards* enhances the _indexing_ performance and allows to
#    _distribute_ a big index across machines.
# 2. Having more *replicas* enhances the _search_ performance and improves the
#    cluster _availability_.
#
# The "number_of_shards" is a one-time setting for an index.
#
# The "number_of_replicas" can be increased or decreased anytime,
# by using the Index Update Settings API.
#
# Elasticsearch takes care about load balancing, relocating, gathering the
# results from nodes, etc. Experiment with different settings to fine-tune
# your setup.

# Use the Index Status API (<http://localhost:9200/A/_status>) to inspect
# the index status.


#################################### Paths ####################################

# Path to directory containing configuration (this file and logging.yml):
#
#path.conf: /path/to/conf

# Path to directory where to store index data allocated for this node.
#
#path.data: /path/to/data
#
# Can optionally include more than one location, causing data to be striped across
# the locations (a la RAID 0) on a file level, favouring locations with most free
# space on creation. For example:
#
#path.data: /path/to/data1,/path/to/data2
path.data: data

# Path to temporary files:
#
#path.work: /path/to/work

# Path to log files:
#
```

```
#path.logs: /path/to/logs

# Path to where plugins are installed:
#
#path.plugins: /path/to/plugins
path.plugins: ${karaf.home}/elasticsearch/plugins

#################################### Plugin ###################################

# If a plugin listed here is not installed for current node, the node will not start.
#
#plugin.mandatory: mapper-attachments,lang-groovy


################################### Memory ####################################

# Elasticsearch performs poorly when JVM starts swapping: you should ensure that
# it _never_ swaps.
#
# Set this property to true to lock the memory:
#
#bootstrap.mlockall: true

# Make sure that the ES_MIN_MEM and ES_MAX_MEM environment variables are set
# to the same value, and that the machine has enough memory to allocate
# for Elasticsearch, leaving enough memory for the operating system itself.
#
# You should also make sure that the Elasticsearch process is allowed to lock
# the memory, eg. by using `ulimit -l unlimited`.


############################## Network And HTTP ###############################

# Elasticsearch, by default, binds itself to the 0.0.0.0 address, and listens
# on port [9200-9300] for HTTP traffic and on port [9300-9400] for node-to-node
# communication. (the range means that if the port is busy, it will automatically
# try the next port).

# Set the bind address specifically (IPv4 or IPv6):
#
#network.bind_host: 192.168.0.1

# Set the address other nodes will use to communicate with this node. If not
# set, it is automatically derived. It must point to an actual IP address.
#
#network.publish_host: 192.168.0.1

# Set both 'bind_host' and 'publish_host':
#
```

```
#network.host: 192.168.0.1
network.host: 127.0.0.1

# Set a custom port for the node to node communication (9300 by default):
#
#transport.tcp.port: 9300

# Enable compression for all communication between nodes (disabled by default):
#
#transport.tcp.compress: true

# Set a custom port to listen for HTTP traffic:
#
#http.port: 9200

# Set a custom allowed content length:
#
#http.max_content_length: 100mb

# Enable HTTP:
#
http.enabled: true
http.cors.enabled: true
http.cors.allow-origin: /.*/


################################### Gateway ###################################

# The gateway allows for persisting the cluster state between full cluster
# restarts. Every change to the state (such as adding an index) will be stored
# in the gateway, and when the cluster starts up for the first time,
# it will read its state from the gateway.

# There are several types of gateway implementations. For more information, see
# <http://elasticsearch.org/guide/en/elasticsearch/reference/current/
modules-gateway.html>.

# The default gateway type is the "local" gateway (recommended):
#
#gateway.type: local

# Settings below control how and when to start the initial recovery process on
# a full cluster restart (to reuse as much local data as possible when using shared
# gateway).

# Allow recovery process after N nodes in a cluster are up:
#
#gateway.recover_after_nodes: 1
```

```
# Set the timeout to initiate the recovery process, once the N nodes
# from previous setting are up (accepts time value):
#
#gateway.recover_after_time: 5m

# Set how many nodes are expected in this cluster. Once these N nodes
# are up (and recover_after_nodes is met), begin recovery process immediately
# (without waiting for recover_after_time to expire):
#
#gateway.expected_nodes: 2


############################# Recovery Throttling #############################

# These settings allow to control the process of shards allocation between
# nodes during initial recovery, replica allocation, rebalancing,
# or when adding and removing nodes.

# Set the number of concurrent recoveries happening on a node:
#
# 1. During the initial recovery
#
#cluster.routing.allocation.node_initial_primaries_recoveries: 4
#
# 2. During adding/removing nodes, rebalancing, etc
#
#cluster.routing.allocation.node_concurrent_recoveries: 2

# Set to throttle throughput when recovering (eg. 100mb, by default 20mb):
#
#indices.recovery.max_bytes_per_sec: 20mb

# Set to limit the number of open concurrent streams when
# recovering a shard from a peer:
#
#indices.recovery.concurrent_streams: 5


################################## Discovery ##################################

# Discovery infrastructure ensures nodes can be found within a cluster
# and master node is elected. Multicast discovery is the default.

# Set to ensure a node sees N other master eligible nodes to be considered
# operational within the cluster. This should be set to a quorum/majority of
# the master-eligible nodes in the cluster.
#
#discovery.zen.minimum_master_nodes: 1
```

```
# Set the time to wait for ping responses from other nodes when discovering.
# Set this option to a higher value on a slow or congested network
# to minimize discovery failures:
#
#discovery.zen.ping.timeout: 3s

# For more information, see
# <http://elasticsearch.org/guide/en/elasticsearch/reference/current/
modules-discovery-zen.html>

# Unicast discovery allows to explicitly control which nodes will be used
# to discover the cluster. It can be used when multicast is not present,
# or to restrict the cluster communication-wise.
#
# 1. Disable multicast discovery (enabled by default):
#
#discovery.zen.ping.multicast.enabled: false
#
# 2. Configure an initial list of master nodes in the cluster
#    to perform discovery when new nodes (master or data) are started:
#
#discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]

# EC2 discovery allows to use AWS EC2 API in order to perform discovery.
#
# You have to install the cloud-aws plugin for enabling the EC2 discovery.
#
# For more information, see
# <http://elasticsearch.org/guide/en/elasticsearch/reference/current/
modules-discovery-ec2.html>
#
# See <http://elasticsearch.org/tutorials/elasticsearch-on-ec2/>
# for a step-by-step tutorial.

# GCE discovery allows to use Google Compute Engine API in order to perform discovery.
#
# You have to install the cloud-gce plugin for enabling the GCE discovery.
#
# For more information, see <https://github.com/elasticsearch/elasticsearch-cloud-gce>.

# Azure discovery allows to use Azure API in order to perform discovery.
#
# You have to install the cloud-azure plugin for enabling the Azure discovery.
#
# For more information, see <https://github.com/elasticsearch/elasticsearch-cloud-azure>.

################################## Slow Log ##################################

# Shard level query and fetch threshold logging.
```

```
#index.search.slowlog.threshold.query.warn: 10s
#index.search.slowlog.threshold.query.info: 5s
#index.search.slowlog.threshold.query.debug: 2s
#index.search.slowlog.threshold.query.trace: 500ms

#index.search.slowlog.threshold.fetch.warn: 1s
#index.search.slowlog.threshold.fetch.info: 800ms
#index.search.slowlog.threshold.fetch.debug: 500ms
#index.search.slowlog.threshold.fetch.trace: 200ms

#index.indexing.slowlog.threshold.index.warn: 10s
#index.indexing.slowlog.threshold.index.info: 5s
#index.indexing.slowlog.threshold.index.debug: 2s
#index.indexing.slowlog.threshold.index.trace: 500ms

################################## GC Logging ################################

#monitor.jvm.gc.young.warn: 1000ms
#monitor.jvm.gc.young.info: 700ms
#monitor.jvm.gc.young.debug: 400ms

#monitor.jvm.gc.old.warn: 10s
#monitor.jvm.gc.old.info: 5s
#monitor.jvm.gc.old.debug: 2s

################################## Security ################################

# Uncomment if you want to enable JSONP as a valid return transport on the
# http server. With this enabled, it may pose a security risk, so disabling
# it unless you need it is recommended (it is disabled by default).
#
#http.jsonp.enable: true
```

It's a "standard" elasticsearch configuration file, allowing you to configure the embedded elasticsearch instance.

Warning: if you change the `etc/elasticsearch.yml` file, you have to restart (with the `bundle:restart` command) the Decanter elasticsearch bundle in order to load the changes.

The Decanter elasticsearch node also supports loading and override of the settings using a `etc/org.apache.karaf.decanter.elasticsearch.cfg` configuration file. This file is not provided by default, as it's used for override of the default settings.

You can override the following elasticsearch properties in this configuration file:

- `cluster.name`
- `http.enabled`
- `node.data`
- `node.name`
- `node.master`
- `path.data`
- `network.host`
- `cluster.routing.schedule`
- `path.plugins`
- `http.cors.enabled`
- `http.cors.allow-origin`

The advantage of using this file is that the elasticsearch node is automatically restarted in order to reload the settings as soon as you change the cfg file.

3.2.2. Embedding Decanter Kibana

In addition of the embedded elasticsearch instance, Decanter also provides an embedded Kibana instance, containing ready to use Decanter dashboards.

The `kibana` feature installs the embedded kibana instance:

```
karaf@root()> feature:install kibana
```

By default, the kibana instance is available on `http://host:8181/kibana`.

The Decanter Kibana instance provides ready to use dashboards:

- Karaf dashboard uses the data harvested by the default JMX collector, and the log collector. Especially, it provides details about the threads, memory, garbage collection, etc.
- Camel dashboard uses the data harvested by the default JMX collector, or the Camel (JMX) collector. It can also leverage the Camel Tracer collector. It provides details about routes processing time, the failed exchanges, etc. This dashboard requires some tuning (updating the queries to match the route IDs).

- ActiveMQ dashboard uses the data harvested by the default JMX collector, or the ActiveMQ (JMX) collector. It provides details about the pending queue, the system usage, etc.
- OperatingSystem dashboard uses the data harvested by the system collector. The default dashboard expects data containing the filesystem usage, and temperature data. It's just a sample, you have to tune the system collector and adapt this dashboard accordingly.

You can change these dashboards to add new panels, change the existing panels, etc.

Of course, you can create your own dashboards, starting from blank or simple dashboards.

By default, Decanter Kibana uses embedded elasticsearch instance. However, it's possible to use a remote elasticsearch instance by providing the elasticsearch parameter on the URL like this for instance:

```
http://localhost:8181/kibana?elasticsearch=http://localhost:9400
```

3.2.3. Elasticsearch Head console

In addition of the embedded elasticsearch instance, Decanter also provides a web console allowing you to monitor and manage your elasticsearch cluster. It's a ready to use elastisearch-head console, directly embedded in Karaf.

The `elasticsearch-head` feature installs the embedded elasticsearch-head web console:

```
karaf@root()> feature:install elasticsearch-head
```

By default, the elasticsearch-head web console is available on `http://host:8181/elasticsearch-head`.

3.3. JDBC

The Decanter JDBC appender allows your to store the data (coming from the collectors) into a database.

The Decanter JDBC appender transforms the data as a json string. The appender stores the json string and the timestamp into the database.

The `decanter-appender-jdbc` feature installs the jdbc appender:

```
karaf@root()> feature:install decanter-appender-jdbc
```

This feature also installs the `etc/org.apache.karaf.decanter.appender.jdbc.cfg` configuration file:

```
#######################################
# Decanter JDBC Appender Configuration
#######################################

# Name of the JDBC datasource
datasource.name=jdbc/decanter

# Name of the table storing the collected data
table.name=decanter

# Dialect (type of the database)
# The dialect is used to create the table
# Supported dialects are: generic, derby, mysql
# Instead of letting Decanter created the table, you can create the table by your own
dialect=generic
```

This configuration file allows you to specify the connection to the database:

- the `datasource.name` property contains the name of the JDBC datasource to use to connect to the database. You can create this datasource using the Karaf `jdbc:create` command (provided by the `jdbc` feature).
- the `table.name` property contains the table name in the database. The Decanter JDBC appender automatically creates the table for you, but you can create the table by yourself. The table is simple and contains just two column:
  - timestamp as INTEGER
  - content as VARCHAR or CLOB
- the `dialect` property allows you to specify the database type (generic, mysql, derby). This property is only used for the table creation.

3.4. JMS

The Decanter JMS appender "forwards" the data (collected by the collectors) to a JMS broker.

The appender sends a JMS Map message to the broker. The Map message contains the harvested data.

The `decanter-appender-jms` feature installs the JMS appender:

```
karaf@root()> feature:install decanter-appender-jms
```

This feature also installs the `etc/org.apache.karaf.decanter.appender.jms.cfg` configuration file containing:

```
#####################################
# Decanter JMS Appender Configuration
#####################################

# Name of the JMS connection factory
connection.factory.name=jms/decanter

# Name of the destination
destination.name=decanter

# Type of the destination (queue or topic)
destination.type=queue

# Connection username
# username=

# Connection password
# password=
```

This configuration file allows you to specify the connection properties to the JMS broker:

- the `connection.factory.name` property specifies the JMS connection factory to use. You can create this JMS connection factory using the `jms:create` command (provided by the `jms` feature).
- the `destination.name` property specifies the JMS destination name where to send the data.
- the `destination.type` property specifies the JMS destination type (queue or topic).
- the `username` property is optional and specifies the username to connect to the destination.
- the `password` property is optional and specifies the username to connect to the destination.

3.5. Camel

The Decanter Camel appender sends the data (collected by the collectors) to a Camel endpoint.

It’s a very flexible appender, allowing you to use any Camel route to transform and forward the harvested data.

The Camel appender creates a Camel exchange and set the "in" message body with a Map of the harvested data. The exchange is send to a Camel endpoint.

The `decanter-appender-camel` feature installs the Camel appender:

```
karaf@root()> feature:install decanter-appender-camel
```

This feature also installs the `etc/org.apache.karaf.decanter.appender.camel.cfg` configuration file containing:

```
#
# Decanter Camel appender configuration
#

# The destination.uri contains the URI of the Camel endpoint
# where Decanter sends the collected data
destination.uri=direct-vm:decanter
```

This file allows you to specify the Camel endpoint where to send the data:

- the `destination.uri` property specifies the URI of the Camel endpoint where to send the data.

The Camel appender send an exchange. The "in" message body contains a Map of the harvested data.

For instance, in this configuration file, you can specify:

```
destination.uri=direct-vm:decanter
```

And you can deploy the following Camel route definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route id="decanter">
      <from uri="direct-vm:decanter"/>
      ...
      ANYTHING
      ...
    </route>
  </camelContext>

</blueprint>
```

This route will receive the Map of harvested data. Using the body of the "in" message, you can do what you want:

- transform and convert to another data format
- use any Camel EIPs (Enterprise Integration Patterns)
- send to any Camel endpoint

4. SLA (Service Level Agreement)

Decanter provides a SLA (Service Level Agreement) layer. It allows you to check values of harvested data (coming from the collectors) and send alerts when the data is not in the expected state.

4.1. Checker

The SLA checker is automatically installed as soon as you install a SLA alerter feature.

It uses the `etc/org.apache.karaf.decanter.sla.checker.cfg` configuration file.

This file contains the check to perform on the collected properties.

The format of this file is:

```
type.propertyName.alertLevel=checkType:value
```

where:

- `type` is optional. It allows you to filter the SLA check for a given type of collected data. It's particulary interesting when Decanter collects multiple JMX object names or servers. You may want to perform different checks depending of the type or source of the collected data.
- `propertyName` is the data property key. For instance, `loggerName`, `message`, `HeapMemoryUsage.used`, etc.
- `alertLevel` is the alerting level for this check. The only two possible values are `error` (critical alert), or `warn` (severe alert).
- `checkType` is the check type. Possible values are `range`, `equal`, `notequal`, `match`, and `notmatch`.
- `value` is the check value, where the data property value has to verify.

The Decanter SLA Checker supports numeric or string check.

To verify a numeric value, you can use:

- `range` to check if the metric is between two values
- `equal` to check if the metric is equal to a value
- `notequal` to check if the metric is not equal to a value

For instance, if you want to check that the number of threads is between 0 and 70, you can use:

```
ThreadCount.error=range:[0,70]
```

You can also filter and specify the type on which we check:

```
jmx-local.ThreadCount.error=range:[0,70]
```

If the thread count is out of this range, Decanter will create an error alert sent to the alerters.

Another example is if you want to check if the myValue is equal to 10:

```
myValue.warn=equal:10
```

If myValue is not equal to 10, Decanter will create a warn alert send to the alerters.

To verify a string value, you can use:

- `match` to check if the metric matches a regex
- `notmatch` to check if the matric doesn’t match a regex

For instance, if you want to create an alert when an ERROR log message happens, you can use:

```
loggerLevel.error=match:ERROR
```

You can also use "complex" regex:

```
loggerName.warn=match:(.*)my\.loggger\.name\.(.*)
```

4.2. Alerters

When the value doesn’t verify the check in the checker configuration, an alert is created an sent to the alerters.

Apache Karaf Decanter provides ready to use alerters.

4.2.1. Log

The Decanter SLA Log alerter log a message for each alert.

The `decanter-sla-log` feature installs the SLA log alerter:

```
karaf@root()> feature:install decanter-sla-log
```

This alerter doesn’t need any configuration.

4.2.2. E-mail

The Decanter SLA e-mail alerter sends an e-mail for each alert.

The `decanter-sla-email` feature installs the SLA e-mail alerter:

```
karaf@root()> feature:install decanter-sla-email
```

This feature also installs the `etc/org.apache.karaf.decanter.sla.email.cfg` configuration file where you can specify the SMTP server and e-mail addresses to use:

```
#
# Decanter SLA e-mail alerter configuration
#

# From e-mail address
from=

# To e-mail address
to=

# Hostname of the SMTP server
host=smtp.gmail.com

# Port of the SMTP server
port=587

# enable SMTP auth
auth=true

# enable starttls and ssl
starttls=true
ssl=false

# Optionally, username for the SMTP server
#username=

# Optionally, password for the SMTP server
#password=
```

- the `from` property specifies the from e-mail address (for instance dev@karaf.apache.org)
- the `to` property specifies the to e-mail address (for instance dev@karaf.apache.org)
- the `host` property specifies the SMTP server hostname or IP address
- the `port` property specifies the SMTP server port number
- the `auth` property (true or false) specifies if the SMTP server requires authentication (true) or not (false)
- the `starttls` property (true or false) specifies if the SMTP server requires STARTTLS (true) or not (false)
- the `ssl` property (true or false) specifies if the SMTP server requires SSL (true) or not (false)

- the `username` property is optional and specifies the username to connect to the SMTP server
- the `password` property is optional and specifies the password to connect to the SMTP server

4.2.3. Camel

The Decanter SLA Camel alerter sends each alert to a Camel endpoint.

It allows you to create a route which reacts to each alert. It’s a very flexible alerter as you can apply transformation, use EIPs, Camel endpoints, etc.

This alerter creates a Camel exchange. The body of the "in" message contains a Map with all alert details (including `alertLevel`, `alertAttribute`, `alertPattern` and all other details).

The `decanter-sla-camel` feature installs the Camel alerter:

```
karaf@root()> feature:install decanter-sla-camel
```

This feature also installs the `etc/org.apache.karaf.decanter.sla.camel.cfg` configuration file:

```
#
# Decanter SLA Camel alerter
#

# alert.destination.uri defines the Camel endpoint URI where
# Decanter send the SLA alerts
alert.destination.uri=direct-vm:decanter-alert
```

This configuration file allows you to specify the Camel endpoint URI where to send the alert (using the `alert.destination.uri` property).

For instance, in this configuration, if you define:

```
alert.destination.uri=direct-vm:decanter-alert
```

You can create the following Camel route which will react to the alert:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route id="decanter-alert">
      <from uri="direct-vm:decanter-alert"/>
      ...
      ANYTHING
      ...
    </route>
  </camelContext>

</blueprint>
```

Developer Guide

1. Architecture

Apache Karaf Decanter uses OSGi EventAdmin to dispatch the harvested data between the collectors and the appenders, and also to throw the alerts to the alerters:

- `decanter/collect/*` EventAdmin topics are used by the collectors to send the harvested data. The appenders consume from these topics and insert the data in a backend.
- `decanter/alert/*` EventAdmin topics are used by the SLA checker to send the alerts. The SLA alerters consume from these topics.

Decanter uses EventAdmin topics as monitoring events dispatcher.

Collectors, appenders, and alerters are simple OSGi services exposed by different bundles.

It means that you can easily extend Decanter adding your own collectors, appenders, or alerters.

2. Custom Collector

A Decanter collector sends an OSGi EventAdmin event to a `decanter/collect/*` topic.

You can create two kinds of collector:

- event driven collector automatically reacts to some internal events. It creates an event sent to a topic.

- polled collector is a Runnable OSGi service periodically executed by the Decanter Scheduler.

2.1. Event Driven Collector

For instance, the log collector is event driven: it automatically reacts to internal log events.

To illustrate an Event Driven Collector, we can create a BundleCollector. This collector will react when a bundle state changes (installed, started, stopped, uninstalled).

The purpose is to send a monitoring event in a collect topic. This monitoring event can be consumed by the appenders.

We create the following `BundleCollector` class implemetings `SynchronousBundleListener` interface:

```
package org.apache.karaf.decanter.sample.collector;

import org.osgi.framework.SynchronousBundleListener;
import org.osgi.service.event.EventAdmin;
import org.osgi.service.event.Event;
import java.util.HashMap;

public class BundleCollector implements SynchronousBundleListener {

    private EventAdmin eventAdmin;

    public BundleCollector(Event eventAdmin) {
      this.eventAdmin = eventAdmin;
    }

    @Override
    public void bundleChanged(BundleEvent bundleEvent) {
      HashMap<String, Object> data = new HashMap<>();
      data.put("type", "bundle");
      data.put("change", bundleEvent.getType());
      data.put("id", bundleEvent.getBundle().getId());
      data.put("location", bundleEvent.getBundle().getLocation());
      data.put("symbolicName", bundleEvent.getBundle().getSymbolicName());
      Event event = new Event("decanter/collect/bundle", data);
      eventAdmin.postEvent(event);
    }

}
```

You can see here the usage of the OSGi EventAdmin as dispatcher: the collector creates a data map, and send it to a `decanter/collect/bundle` topic.

We just need an Activator in the collector bundle to start our BundleCollector listener:

```
package org.apache.karaf.decanter.sample.collector;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import org.osgi.framework.ServiceRegistration;
import org.osgi.service.event.EventAdmin;
import org.osgi.util.tracker.ServiceTracker;

public class Activator implements BundleActivator {

       private BundleCollector collector;

       public void start(final BundleContext bundleContext) {
           ServiceTracker tracker = new ServiceTracker(bundleContext,
EventAdmin.class.getName(), null);
           EventAdmin eventAdmin = (EventAdmin) tracker.waitForService(10000);
           collector = new BundleCollector(eventAdmin);
       }

       public void stop(BundleContext bundleContext) {
           collector = null;
       }

}
```

Now, we just need a Maven `pom.xml` to package the bundle with the correct OSGi headers:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <!--

        Licensed to the Apache Software Foundation (ASF) under one or more
        contributor license agreements.  See the NOTICE file distributed with
        this work for additional information regarding copyright ownership.
        The ASF licenses this file to You under the Apache License, Version 2.0
        (the "License"); you may not use this file except in compliance with
        the License.  You may obtain a copy of the License at

           http://www.apache.org/licenses/LICENSE-2.0

        Unless required by applicable law or agreed to in writing, software
        distributed under the License is distributed on an "AS IS" BASIS,
        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
        See the License for the specific language governing permissions and
        limitations under the License.
    -->

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.apache.karaf.decanter.sample.collector</groupId>
    <artifactId>org.apache.karaf.decanter.sample.collector.bundle</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>bundle</packaging>
    <name>Apache Karaf :: Decanter :: Sample :: Collector :: Bundle</name>

    <dependencies>

        <!-- OSGi -->
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.core</artifactId>
            <version>4.3.1</version>
        </dependency>
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.compendium</artifactId>
            <version>4.3.1</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
```

```
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.4.0</version>
                <inherited>true</inherited>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Bundle-Version>${project.version}</Bundle-Version>

<Bundle-Activator>org.apache.karaf.decanter.sample.collector.bundle.Activator</Bundle-Activator>
                        <Import-Package>
                            *
                        </Import-Package>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

You can now enable this collector, just by installing the bundle in Apache Karaf (using the deploy folder, or the `bundle:install` command.

2.2. Polled Collector

You can also create a polled collector.

A polled collector is basically a Runnable OSGi service, periodically executed for you by the Decanter Scheduler.

The run() method of the polled collector is responsible to harvest the data and send the monitoring event.

For instance, we can create a very simple polled collector sending a constant `Hello World` string.

We create the HelloCollector class implementing the Runnable interface:

```
package org.apache.karaf.decanter.sample.collector.hello;

import org.osgi.service.event.Event;
import org.osgi.service.event.EventAdmin;
import java.util.HashMap;

public class HelloCollector implements Runnable {

  private EventAdmin eventAdmin;

  public HelloCollector(EventAdmin eventAdmin) {
    this.eventAdmin = eventAdmin;
  }

  @Override
  public void run() {
    HashMap<String, Object> data = new HashMap<>();
    data.put("type", "hello");
    data.put("message", "Hello World");
    Event event = new Event("decanter/collect/hello", data);
    eventAdmin.postEvent(event);
  }

}
```

You can see the `run()` method which post the monitoring event in the `decanter/collector/hello` topic.

We just need a BundleActivator to register the HelloCollector as an OSGi service:

```
package org.apache.karaf.decanter.sample.collector.hello;

import org.osgi.framework.*;
import org.osgi.service.event.EventAdmin;
import org.osgi.util.tracker.ServiceTracker;

public class Activator implements BundleActivator {

    private ServiceRegistration registration;

    public void start(BundleContext bundleContext) {
       ServiceTracker tracker = new ServiceTracker(bundleContext,
EventAdmin.class.getName(), null);
       EventAdmin eventAdmin = tracker.waitForService(10000);
       HelloCollector collector = new HelloCollector(eventAdmin);

       Dictionary<String, String> serviceProperties = new Hashtable<String, String>();
       serviceProperties.put("decanter.collector.name", "hello");
       registration = bundleContext.registerService(Runnable.class, collector,
serviceProperties);
    }

    public void stop(BundleContext bundleContext) {
       if (registration != null) registration.unregister();
    }

}
```

Now, we can package the bundle using the following Maven pom.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <!--

        Licensed to the Apache Software Foundation (ASF) under one or more
        contributor license agreements.  See the NOTICE file distributed with
        this work for additional information regarding copyright ownership.
        The ASF licenses this file to You under the Apache License, Version 2.0
        (the "License"); you may not use this file except in compliance with
        the License.  You may obtain a copy of the License at

           http://www.apache.org/licenses/LICENSE-2.0

        Unless required by applicable law or agreed to in writing, software
        distributed under the License is distributed on an "AS IS" BASIS,
        WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
        See the License for the specific language governing permissions and
        limitations under the License.
    -->

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.apache.karaf.decanter.sample.collector</groupId>
    <artifactId>org.apache.karaf.decanter.sample.collector.hello</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>bundle</packaging>
    <name>Apache Karaf :: Decanter :: Sample :: Collector :: Hello</name>

    <dependencies>

        <!-- OSGi -->
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.core</artifactId>
            <version>4.3.1</version>
        </dependency>
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.compendium</artifactId>
            <version>4.3.1</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
```

```
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.4.0</version>
                <inherited>true</inherited>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Bundle-Version>${project.version}</Bundle-Version>

<Bundle-Activator>org.apache.karaf.decanter.sample.collector.hello.Activator</Bundle-Activator>
                        <Import-Package>
                            *
                        </Import-Package>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

You can now enable this collector, just by installing the bundle in Apache Karaf (using the deploy folder, or the `bundle:install` command.

3. Custom Appender

A Decanter Appender is an OSGi EventAdmin EventHandler: it's listening of `decanter/collect/*` EventAdmin topics, and receives the monitoring data coming from the collectors.

It's responsible to store the data into a target backend.

To enable a new Decanter Appender, you just have to register an EventHandler OSGi service.

For instance, if you want to create a very simple SystemOutAppender that displays the monitoring data (coming from the collectors) to System.out, you can create the following SystemOutAppender class implementing EventHandler interface:

```
package org.apache.karaf.decanter.sample.appender.systemout;

import org.osgi.service.event.Event;
import org.osgi.service.event.EventHandler;

import java.util.HashMap;

public class SystemOutAppender implements EventHandler {

    @Override
    public void handleEvent(Event event) {
        for (String name : event.getPropertyNames()) {
            System.out.println(name + ":" + event.getProperty(name));
        }
    }

}
```

Now, we create a BundleActivator that register our SystemOutAppender as an EventHandler OSGi service:

```
package org.apache.karaf.decanter.sample.appender.systemout;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceRegistration;
import org.osgi.service.event.EventConstants;
import org.osgi.service.event.EventHandler;
import java.util.HashMap;
import java.util.Dictionary;

public class Activator implements BundleActivator {

  private ServiceRegistration registration;

  public void start(BundleContext bundleContext) {
    SystemOutAppender appender = new SystemOutAppender();
    Dictionary<String, String> properties = new Hashtable<>();
    properties.put(EventConstants.EVENT_TOPIC, "decanter/collect/*");
    registration =  bundleContext.registerService(EventHandler.class, appender,
properties);
  }

  public void stop(BundleContext bundleContext) {
    if (registration != null) registration.unregister();
  }

}
```

You can see that our SystemOutAppender will listen on any `decanter/collect/*` topics.

We can now package our appender bundle using the following Maven `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">)

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.apache.karaf.decanter.sample.appender</groupId>
    <artifactId>org.apache.karaf.decanter.sample.appender.systemout</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>bundle</packaging>
    <name>Apache Karaf :: Decanter :: Sample :: Appender :: SystemOut</name>

    <dependencies>

        <!-- OSGi -->
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.core</artifactId>
            <version>4.3.1</version>
        </dependency>
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.compendium</artifactId>
            <version>4.3.1</version>
        </dependency>


    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.4.0</version>
                <inherited>true</inherited>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Bundle-Version>${project.version}</Bundle-Version>

<Bundle-Activator>org.apache.karaf.decanter.sample.appender.systemout.Activator</Bundle-Activator>
                        <Import-Package>
                            *
                        </Import-Package>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Once built, you can enable this appender by deploying the bundle in Karaf (using the deploy folder or the `bundle:install` command).

4. Custom SLA Alerter

A Decanter SLA Alerter is basically a special kind of appender.

It's an OSGi EventAdmin EventHandler: it's listening of `decanter/alert/*` EventAdmin topics, and receives the alerting data coming from the SLA checker.

To enable a new Decanter Alerter, you just have to register an EventHandler OSGi service, like we do for an appender.

For instance, if you want to create a very simple SystemOutAlerter that displays the alert (coming from the SLA checker) to System.out, you can create the following SystemOutAlerter class implementing EventHandler interface:

```
package org.apache.karaf.decanter.sample.alerter.systemout;

import org.osgi.service.event.Event;
import org.osgi.service.event.EventHandler;

import java.util.HashMap;

public class SystemOutAlerter implements EventHandler {

    @Override
    public void handleEvent(Event event) {
        for (String name : event.getPropertyNames()) {
            System.err.println(name + ":" + event.getProperty(name));
        }
    }

}
```

Now, we create a BundleActivator that register our SystemOutAppender as an EventHandler OSGi service:

```
package org.apache.karaf.decanter.sample.alerter.systemout;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.Constants;
import org.osgi.framework.ServiceRegistration;
import org.osgi.service.event.EventConstants;
import org.osgi.service.event.EventHandler;
import java.util.HashMap;
import java.util.Dictionary;

public class Activator implements BundleActivator {

  private ServiceRegistration registration;

  public void start(BundleContext bundleContext) {
    SystemOutAlerter alerter = new SystemOutAlerter();
    Dictionary<String, String> properties = new Hashtable<>();
    properties.put(EventConstants.EVENT_TOPIC, "decanter/alert/*");
    registration =  bundleContext.registerService(EventHandler.class, alerter,
properties);
  }

  public void stop(BundleContext bundleContext) {
    if (registration != null) registration.unregister();
  }

}
```

You can see that our SystemOutAlerter will listen on any `decanter/alert/*` topics.

We can now package our alerter bundle using the following Maven `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">)

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.apache.karaf.decanter.sample.alerter</groupId>
    <artifactId>org.apache.karaf.decanter.sample.alerter.systemout</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>bundle</packaging>
    <name>Apache Karaf :: Decanter :: Sample :: Alerter :: SystemOut</name>

    <dependencies>

        <!-- OSGi -->
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.core</artifactId>
            <version>4.3.1</version>
        </dependency>
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.compendium</artifactId>
            <version>4.3.1</version>
        </dependency>


    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.4.0</version>
                <inherited>true</inherited>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Bundle-Version>${project.version}</Bundle-Version>

<Bundle-Activator>org.apache.karaf.decanter.sample.alerter.systemout.Activator</Bundle-Activator>
                        <Import-Package>
                            *
                        </Import-Package>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Once built, you can enable this alerter by deploying the bundle in Karaf (using the deploy folder or the `bundle:install` command).

Last updated 2015-08-02 08:44:39 CEST