APACHE KARAF DECANTER 2.X - DOCUMENTATION

Apache Software Foundation

APACHE KARAF DECANTER 2.X - DOCUMENTATION

1. USER GUIDE

1.1. INTRODUCTION

Apache Karaf Decanter is a monitoring solution running in Apache Karaf.

It's composed of three parts:

- Collectors are responsible for harvesting monitoring data. Decanter provides collectors to harvest different kinds 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 on the backend storage that you want to use.
- Alerters are a special kind of appender. A check is performed on all harvested data. If a check fails, an alert event is created and sent to the alerters. Decanter provides alerters depending on the kind of notification that you want.

Apache Karaf Decanter provides Karaf features for each collector, appender, 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/2.0.0/xml/features
```

Or

```
karaf@root()> feature:repo-add decanter 2.0.0
```

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

1.2. 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.

1.2.1. LOG

The Decanter Log Collector is an event driven collector. It automatically reacts when a log occurs, and sends 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.

The Decanter log collector is using osgi:DecanterLogCollectorAppender appender. In order to work, your Apache Karaf Pax Logging configuration should contain this appender.

The default Apache Karaf etc/org.ops4j.pax.logging.cfg configuration file is already fine:

```
log4j.rootLogger = DEBUG, out, osgi:*
```

If you want, you can explicitly specify the DecanterLogCollectorAppender appender:

```
log4j.rootLogger = DEBUG, out, osgi:DecanterLogCollectorAppender,
osgi:VmLogAppender
```

1.2.2. CXF LOGGING FEATURE INTEGRATION

The CXF message logging nicely integrates with Decanter. Simply add the org.apache.cxf.ext.logging.LoggingFeature to your service.

This will automatically log the messages from all clients and endpoints to slf4j. All meta data can be found in the MDC attributes. The message logging can be switched on/off per service using the org.ops4j.pax.logging.cfg.

When using with Decanter make sure you enable the log collector to actually process the message logs.

1.2.3. LOG SOCKET

The Decanter Log Socket Collector is an event driven collector. It creates a socket, waiting for incoming event. The expected events are log4j LoggingEvent. The log4j LoggingEvent is transformed as a Map containing the log details (level, logger name, message, ...). This Map is sent to the appenders.

The collector allows you to remotely use Decanter. For instance, you can have an application running on a different platform (spring-boot, application servers, ...). This application can use a log4j socket appender that send the logging events to the Decanter log socket collector.

The decanter-collector-log-socket feature install the log socket collector:

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

This feature installs the collector and a default etc/org.apache.karaf.decanter.collector.log.socket.cfg configuration file containing:

```
#
# Decanter Log/Log4j Socket collector configuration
#

#port=4560
#workers=10
```

- the port property defines the port number where the collector is bound and listen for incoming logging event. Default is 4560.
- the workers properties defines the number of threads (workers) which can deal with multiple clients in the same time.

1.2.4. 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 to local Karaf log messages, whereas the file collector can react to any file, including log files from other systems to Karaf. It means that you can monitor and send collected data for any system (even if it is not Java based).

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 monitor
- 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.

You can also filter the lines read from the file using the regex property.

For instance:

```
regex=(.*foo.*)
```

Only the line matching the regex will be sent to the dispatcher.

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
regex=(.*my.*)
my=stuff
```

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

PARSER

By default, the collector use the org.apache.karaf.decanter.impl.parser.IdentityParser parser to parse the line into a typed Object (Long, Integer or String) before send it to the EventDispatcher data map.

IDENTITY PARSER

The identity parser doesn't actually parse the line, it just passes through. It's the default parser used by the file collector.

SPLIT PARSER

The split parser splits the line using a separator (, by default). Optionally, it can take keys used a property name in the event.

For instance, you can have the following etc/org.apache.karaf.decanter.parser.split.cfg configuration file:

```
separator=,
keys=first,second,third,fourth
```

If the parser gets a line (collected by the file collector) like this,is,a,test, the line will be parsed as follows (the file collector will send the following data to the dispatcher):

```
first->this
second->is
third->a
fourth->test
```

If the keys configuration is not set, then key-0, key-1, etc will be used.

To use this parser in the file collector, you have to define it in the parser.target configuration (in etc/org.apache.karaf.decanter.collector.file-XXXX.cfg):

```
parser.target=(parserId=split)
```

REGEX PARSER

The regex parser is similar to the split parser but instead of using a separator, it uses regex groups.

The configuration contains the regex and the keys in the etc/org.apache.karaf.decanter.parser.regex.cfg configuration file:

```
regex=(t.*t)
```

If the parser gets a line (collected by the file collector) like a test here, the line will be parsed as follows (the file collector will send the following data to the dispatcher):

```
key-0->test
```

It's also possible to use keys to identify each regex group.

To use this parser in the file collector, you have to define it in the parser.target configuration (in etc/org.apache.karaf.decanter.collector.file-XXXX.cfg):

```
parser.target=(parserId=regex)
```

CUSTOM PARSER

You can write your own parser by implementing the org.apache.karaf.decanter.api.parser.Parser interface and declare it into the file collector configuration file:

```
parser.target=(parserId=myParser)
```

1.2.5. EVENTADMIN

The Decanter EventAdmin Collector is an event-driven collector, listening for all internal events happening in the Apache Karaf Container.

It's the perfect way to audit all actions performed on resources (features, bundles, configurations, ...) by users (via local shell console, SSH, or JMX).

We recommend to use this collector to implement users and actions auditing.

The decanter-collector-eventadmin feature installs the eventadmin collector:

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

By default, the eventadmin collector is listening for all OSGi framework and Karaf internal events.

You can specify additional events to trap by providing a 'etc/org.apache.karaf.decanter.collector.eventadmin-my.cfg' configuration file, containing the EventAdmin topics you want to listen:

```
event.topics=my/*
```

By default, the events contain timestamp and subject. You can disable this by modifying etc/org.apache.felix.eventadmin.impl.EventAdmin configuration file:

```
org.apache.felix.eventadmin.AddTimestamp=true
org.apache.felix.eventadmin.AddSubject=true
```

1.2.6. 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.

In addition, the JMX collector also supports the execution of operations on dedicated ObjectName that

you configure via cfg file.

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=*

# Several object names can also be specified.
# What matters is that the property names begin with "object.name".
#object.name.system=java.lang:*
#object.name.karaf=org.apache.karaf:type=http,name=*
#object.name.3=org.apache.activemq:*

# You can also execute operations on some MBeans, providing the object name, operation,
arguments (separated by ,)
# and signatures (separated by ,) for the arguments (separated by |)
#operation.name.rootLogger=org.apache.karaf:type=log,name=root|getLevel|rootLogger|java.lang.
String
```

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 to. "local" is a 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 prefix 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 of some MBeans. Several object names can be listed, provided the property prefix is object.name..
- 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.
- the operation.name prefix is also optional. You can use it to execute an operation. The value format is objectName|operation|arguments|signatures.

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.

JMXMP

The Karaf Decanter JMX collector by default uses RMI protocol for JMX. But it also supports JMXMP protocol.

The features to install are the same: decanter-collector-jmx.

However, you have to enable the jmxmp protocol support in the Apache Karaf instance hosting Karaf Decanter.

You can download the jmxmp protocol provider artifact on Maven Central: [http://repo.maven.apache.org/maven2/org/glassfish/external/opendmk_jmxremote_optional_jar/1.0-b01-ea/opendmk_jmxremote_optional_jar-1.0-b01-ea.jar]

The opendmk_jmxremote_optional_jar-1.0-b01-ea.jar file has to be copied in the lib/boot folder of your Apache Karaf instance.

Then, you have to add the new JMX remote packages by editing etc/config.properties, appending the following to the org.osgi.framework.system.packages.extra property:

```
org.osgi.framework.system.packages.extra = \
   ...
   javax.remote, \
   com.sun.jmx, \
   com.sun.jmx.remote, \
   com.sun.jmx.remote.protocol, \
   com.sun.jmx.remote.generic, \
   com.sun.jmx.remote.opt, \
   com.sun.jmx.remote.opt.internal, \
   com.sun.jmx.remote.opt.security, \
   com.sun.jmx.remote.opt.util, \
   com.sun.jmx.remote.profile, \
   com.sun.jmx.remote.profile.sasl, \
   com.sun.jmx.remote.profile.tls, \
   com.sun.jmx.remote.socket, \
   javax.management, \
   javax.management.remote, \
   javax.management.remote.generic, \
   javax.management.remote.jmxmp, \
   javax.management.remote.message
```

Then, you can create a new Decanter JMX collector by creating a new file, like etc/org.apache.karaf.decanter.collector.jmx-mycollector.cfg containing something like:

```
type=jmx-mycollector
url=service:jmx:jmxmp://host:port
jmx.remote.protocol.provider.pkgs=com.sun.jmx.remote.protocol
```

You can see: * the url property contains an URL with jmxmp instead of rmi. * in order to support jmxmp protocol, you have to set the protocol provider via the jmx.remote.protocol.provider.pkgs property (by default, Karaf Decanter JMX collector uses the rmi protocol provider)

1.2.7. 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-jmx-activemq
```

This feature installs the same collector as the decanter-collector-jmx, but also adds 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.

1.2.8. CAMEL (JMX)

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

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

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

This feature installs the same collector as the decanter-collector-jmx, but also adds 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.

1.2.9. CAMEL TRACER & NOTIFIER

Decanter provides a Camel Tracer Handler that you can set on a Camel Tracer. It also provides a Camel Event Notifier.

CAMEL TRACER

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

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

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

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="dispatcher" interface="org.osgi.service.event.EventAdmin"/>

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

You can extend the Decanter event with any property using a custom DecanterCamelEventExtender:

```
public interface DecanterCamelEventExtender {

  void extend(Map<String, Object> decanterData, Exchange camelExchange);

}
```

You can inject your extender using setExtender(myExtender) on the DecanterTraceEventHandler. Decanter will automatically call your extender to populate extra properties.

CAMEL EVENT NOTIFIER

Decanter also provides DecanterEventNotifier implementing a Camel event notifier: http://camel.apache.org/eventnotifier-to-log-details-about-all-sent-exchanges.html

It's very similar to the Decanter Camel Tracer. You can control the camel contexts and routes to which you want to trap events.

1.2.10. SYSTEM (OSHI)

The oshi collector is a system collector (polled) that periodically retrieve all details about the hardware and the operating system.

This collector gets lot of details about the machine.

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

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

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

```
################################################################################
#
#   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.
#
################################################################################

#
# Decanter oshi (system) collector
#

# computerSystem=true
# computerSystem.baseboard=true
# computerSystem.firmware=true
# memory=true
# processors=true
# processors.logical=true
# displays=true
# disks=true
# disks.partitions=true
# graphicsCards=true
# networkIFs=true
# powerSources=true
# soundCards=true
# sensors=true
# usbDevices=true
# operatingSystem=true
# operatingSystem.fileSystems=true
# operatingSystem.networkParams=true
# operatingSystem.processes=true
# operatingSystem.services=true
```

By default, the oshi collector gets all details about the machine. You can filter what you want to harvest in this configuration file.

1.2.11. SYSTEM (SCRIPT)

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
#
# You can define the number of thread to use for parallelization command calls:
# thread.number=1
#
# The format is command.key=command_to_execute
# where command is a reserved keyword used to identify a command property
# for instance:
#
# command.df=df -h
# command.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:
#
# command.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:

```
command.name=system_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:

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

1.2.12. NETWORK SOCKET

The Decanter network socket collector listens for incoming messages coming from a remote network socket collector.

The decanter-collector-socket feature installs the network socket collector:

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

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

```
# Decanter Socket Collector

# Port number on which to listen
#port=34343

# Number of worker threads to deal with
#workers=10

# Protocol tcp(default) or udp
#protocol=tcp

# Unmarshaller to use
# Unmarshaller is identified by data format. The default is json, but you can use another
unmarshaller
unmarshaller.target=(dataFormat=json)
```

- the port property contains the port number where the network socket collector is listening
- the workers property contains the number of worker threads the socket collector is using for the connection
- the protocol property contains the protocol used by the collector for transferring data with the client
- the unmarshaller.target property contains the unmarshaller used by the collector to transform the data sent by the client.

1.2.13. JMS

The Decanter JMS collector consumes the data from a JMS queue or topic. It's a way to aggregate collected data coming from (several) remote machines.

The decanter-collector-jms feature installs the JMS collector:

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

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

```
########################################
# Decanter JMS Collector 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=
```

- the connection.factory.name is the name of the ConnectionFactory OSGi service to use
- the destination.name is the name of the queue or topic where to consume messages from the JMS broker
- the destination.type is the type of the destination (queue or topic)
- the username and password properties are the credentials to use with a secured connection factory

1.2.14. MQTT

The Decanter MQTT collector receives collected messages from a MQTT broker. It's a way to aggregate collected data coming from (several) remote machines.

The decanter-collector-mqtt feature installs the MQTT collector:

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

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

```
#######################################
# Decanter MQTT Collector Configuration
#######################################

# URI of the MQTT broker
server.uri=tcp://localhost:61616

# MQTT Client ID
client.id=decanter

# MQTT topic name
topic=decanter
```

- the server.uri is the location of the MQTT broker
- the client.id is the Decanter MQTT client ID
- the topic is the MQTT topic pattern where to receive the messages

1.2.15. KAFKA

The Decanter Kafka collector receives collected messages from a Kafka broker. It's a way to aggregate collected data coming from (several) remote machines.

The decanter-collector-kafka feature installs the Kafka collector:

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

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

```
###############################
# Decanter Kafka Configuration
###############################

# A list of host/port pairs to use for establishing the initial connection to the Kafka cluster
#bootstrap.servers=localhost:9092

# An id string to identify the group where the consumer belongs to
#group.id=decanter

# Enable auto commit of consumed messages
#enable.auto.commit=true

# Auto commit interval (in ms) triggering the commit
#auto.commit.interval.ms=1000

# Timeout on the consumer session
```

```
#session.timeout.ms=30000

# Serializer class for key that implements the Serializer interface
#key.serializer=org.apache.kafka.common.serialization.StringSerializer

# Serializer class for value that implements the Serializer interface.
#value.serializer=org.apache.kafka.common.serialization.StringSerializer

# Name of the topic
#topic=decanter

# Security (SSL)
#security.protocol=SSL

# SSL truststore location (Kafka broker) and password
#ssl.truststore.location=${karaf.etc}/keystores/keystore.jks
#ssl.truststore.password=karaf

# SSL keystore (if client authentication is required)
#ssl.keystore.location=${karaf.etc}/keystores/clientstore.jks
#ssl.keystore.password=karaf
#ssl.key.password=karaf

# (Optional) SSL provider (default uses the JVM one)
#ssl.provider=

# (Optional) SSL Cipher suites
#ssl.cipher.suites=

# (Optional) SSL Protocols enabled (default is TLSv1.2,TLSv1.1,TLSv1)
#ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1

# (Optional) SSL Truststore type (default is JKS)
#ssl.truststore.type=JKS

# (Optional) SSL Keystore type (default is JKS)
#ssl.keystore.type=JKS

# Security (SASL)
# For SASL, you have to configure Java System property as explained in
http://kafka.apache.org/documentation.html#security_ssl
```

The configuration is similar to the Decanter Kafka appender. Please, see the Kafka collector for details.

1.2.16. REST SERVLET

The Decanter Rest Servlet collector registers a servlet on the OSGi HTTP service (by default on /decanter/collect).

It listens for incoming collected messages on this servlet.

The decanter-collector-rest-servlet feature installs the collector:

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

1.2.17. SOAP

The Decanter SOAP collector periodically requests a SOAP service and returns the result (the SOAP Response, or error details if it failed).

The decanter-collector-soap feature installs the collector:

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

This feature also installs etc/org.apache.karaf.decanter.collector.soap.cfg configuration file where you can setup the URL of the service and the SOAP request to use:

```
#
# Decanter SOAP collector
#

url=http://localhost:8080/cxf/service
soap.request=
```

The collector sends several collected properties to the dispatcher, especially:

- soap.response property contains the actual SOAP response
- error is only populated when the service request failed, containing the error detail
- http.response.code contains the HTTP status code of the service request

1.2.18. DROPWIZARD METRICS

The Decanter Dropwizard Metrics collector get a MetricSet OSGi service and periodically get the metrics in the set.

The decanter-collector-dropwizard feature installs the collector:

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

As soon as a MetricSet (like MetricRegistry) service will be available, the collector will get the metrics and send to the Decanter dispatcher.

1.2.19. JDBC

The Decanter JDBC collector periodically executes a query on a database and sends the query result to the dispatcher.

The decanter-collector-jdbc feature installs the JDBC collector:

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

The feature also installs the etc/org.apache.karaf.decanter.collector.jdbc.cfg configuration file:

```
########################################
# Decanter JDBC Collector Configuration
########################################

# DataSource to use
dataSource.target=(osgi.jndi.service.name=jdbc/decanter)

# SQL Query to retrieve data
query=select * from TABLE
```

This configuration file allows you to configure the datasource to use and the SQL query to perform:

- 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 query property contains the SQL query to perform on the database and retrieve data.

This collector is periodically executed by the Karaf scheduler.

1.2.20. CONFIGADMIN

The Decanter ConfigAdmin collector listens for any configuration change and send the updated configuration to the dispatcher.

The decanter-collector-configadmin feature installs the ConfigAdmin collector:

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

1.2.21. PROMETHEUS

The Decanter Prometheus collector is able to periodically (scheduled collector) read Prometheus servlet output to create events sent in Decanter.

The decanter-collector-prometheus feature installs the Prometheus collector:

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

The feature also installs the etc/org.apache.karaf.decanter.collector.prometheus.cfg configuration file containing:

```
prometheus.url=http://host/prometheus
```

The prometheus.url property is mandatory and define the location of the Prometheus export servlet (that could be provided by the Decanter Prometheus appender for instance).

1.2.22. REDIS

The Decanter Redis collector is able to periodically (scheduled collector) read Redis Map to get key/value pairs. You can filter the keys you want thanks to key pattern.

The decanter-collector-redis feature installs the Redis collector:

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

The feature also installs the etc/org.apache.karaf.decanter.collector.redis.cfg configuration file containing:

```
address=localhost:6379

#
# Define the connection mode.
# Possible modes: Single (default), Master_Slave, Sentinel, Cluster
#
mode=Single

#
# Name of the Redis map
# Default is Decanter
#
map=Decanter

#
# For Master_Slave mode, we define the location of the master
# Default is localhost:6379
#
#masterAddress=localhost:6379

#
# For Sentinel model, define the name of the master
# Default is myMaster
#
#masterName=myMaster

#
# For Cluster mode, define the scan interval of the nodes in the cluster
# Default value is 2000 (2 seconds).
#
#scanInterval=2000

#
# Key pattern to looking for.
# Default is *
#
#keyPattern=*
```

You can configure the Redis connection (depending of the topology) and the key pattern in this configuration file.

1.2.23. ELASTICSEARCH

The Decanter Elasticsearch collector retrieves documents from Elasticsearch periodically (scheduled collector). By default, it harvests all documents in the given index, but you can also specify a query.

The decanter-collector-elasticsearch feature installs the Elasticsearch collector:

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

The feature also install etc/org.apache.karaf.decanter.collector.elasticsearch.cfg configuration file containing:

```
# HTTP address of the elasticsearch nodes (separated with comma)
addresses=http://localhost:9200

# Basic username and password authentication (no authentication by default)
#username=user
#password=password

# Name of the index to request (decanter by default)
#index=decanter

# Query to request document (match all by default)
#query=

# Starting point for the document query (no from by default)
#from=

# Max number of documents retrieved (no max by default)
#max=

# Search timeout, in seconds (no timeout by default)
#timeout=
```

- addresses property is the location of the Elasticsearch instances. Default is http://localhost:9200.
- username and password properties are used for authentication. They are null (no authentication) by default.
- index property is the Elasticsearch index where to get documents. It's decanter by default.
- query property is a search query to use. Default is null meaning all documents in the index are harvested.
- from and max properties are used to "square" the query. They are null by default.
- timeout property limits the query execution. There's no timeout by default.

1.2.24. CUSTOMIZING PROPERTIES IN COLLECTORS

You can add, rename or remove properties collected by the collectors before sending it to the dispatcher.

In the collector configuration file (for instance etc/org.apache.karaf.decanter.collector.jmx-local.cfg for the local JMX collector), you can add any property. By default, the property is added to the data sent to the dispatcher.

You can prefix the configuration property with the action you can perform before sending:

- fields.add. adds a property to the data sent. The following add property hello with value world:

```
---
fields.add.hello=world
---
```

- fields.remove. removes a property to the data sent:

```
---
fields.remove.hello=
---
```

- fields.rename. rename a property with another name:

```
---
fields.rename.helo=hello
---
```

1.3. APPENDERS

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

1.3.1. 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.

1.3.2. ELASTICSEARCH APPENDER

The Decanter Elasticsearch Appender stores the data (coming from the collectors) into an Elasticsearch instance. It transforms the data as a json document, stored into Elasticsearch.

The Decanter Elasticsearch appender uses the Elasticsearch Rest client provided since Elasticsearch 5.x. It can be used with any Elasticsearch versions.

The decanter-appender-elasticsearch feature installs this appender:

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

You can configure the appender (especially the Elasticsearch location) in etc/org.apache.karaf.decanter.appender.elasticsearch.cfg configuration file.

1.3.3. FILE

The Decanter File appender stores the collected data in a CSV file.

The decanter-appender-file feature installs the file appender:

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

By default, the file appender stores the collected data in ${karaf.data}/decanter file. You can change the file where to store the data using the filename property in etc/org.apache.karaf.decanter.appender.file.cfg configuration file.

You can also change the marshaller to use. By default, the marshaller used is the CSV one. But you can switch to the JSON one using the marshaller.target property in etc/org.apache.karaf.decanter.appender.file.cfg configuration file.

1.3.4. JDBC

The Decanter JDBC appender allows you 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 columns:
  - 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.

1.3.5. 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.

1.3.6. 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 sends 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

1.3.7. KAFKA

The Decanter Kafka appender sends the data (collected by the collectors) to a Kafka topic.

The decanter-appender-kafka feature installs the Kafka appender:

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

This feature installs a default etc/org.apache.karaf.decanter.appender.kafka.cfg configuration file containing:

```
###################################
# Decanter JMS Kafka Configuration
###################################

# A list of host/port pairs to use for establishing the initial connection to the Kafka cluster
#bootstrap.servers=localhost:9092

# An id string to pass to the server when making requests
# client.id

# The compression type for all data generated by the producer
# compression.type=none

# The number of acknowledgments the producer requires the leader to have received before considering a request complete
# - 0: the producer doesn't wait for ack
# - 1: the producer just waits for the leader
```

```
# - all: the producer waits for leader and all followers (replica), most secure
# acks=all

# Setting a value greater than zero will cause the client to resend any record whose send fails with a
potentially transient error
# retries=0

# The producer will attempt to batch records together into fewer requests whenever multiple
records are being sent to the same partition
# batch.size=16384

# The total bytes of memory the producer can use to buffer records waiting to be sent to the server.
# If records are sent faster than they can be delivered to the server the producer will either block or
throw an exception
# buffer.memory=33554432

# Serializer class for key that implements the Serializer interface
# key.serializer=org.apache.kafka.common.serialization.StringSerializer

# Serializer class for value that implements the Serializer interface.
# value.serializer=org.apache.kafka.common.serialization.StringSerializer

# Producer request timeout
# request.timeout.ms=5000

# Max size of the request
# max.request.size=2097152

# Name of the topic
# topic=decanter

# Security (SSL)
# security.protocol=SSL

# SSL truststore location (Kafka broker) and password
# ssl.truststore.location=${karaf.etc}/keystores/keystore.jks
# ssl.truststore.password=karaf

# SSL keystore (if client authentication is required)
# ssl.keystore.location=${karaf.etc}/keystores/clientstore.jks
# ssl.keystore.password=karaf
# ssl.key.password=karaf

# (Optional) SSL provider (default uses the JVM one)
# ssl.provider=

# (Optional) SSL Cipher suites
# ssl.cipher.suites=

# (Optional) SSL Protocols enabled (default is TLSv1.2,TLSv1.1,TLSv1)
# ssl.enabled.protocols=TLSv1.2,TLSv1.1,TLSv1

# (Optional) SSL Truststore type (default is JKS)
# ssl.truststore.type=JKS

# (Optional) SSL Keystore type (default is JKS)
# ssl.keystore.type=JKS
```

```
# Security (SASL)
# For SASL, you have to configure Java System property as explained in
http://kafka.apache.org/documentation.html#security_ssl
```

This file allows you to define how the messages are sent to the Kafka broker:

- the bootstrap.servers contains a list of host:port of the Kafka brokers. Default value is localhost:9092.
- the client.id is optional. It identifies the client on the Kafka broker.
- the compression.type defines if the messages have to be compressed on the Kafka broker. Default value is none meaning no compression.
- the acks defines the acknowledgement policy. Default value is all. Possible values are:
  - 0 means the appender doesn't wait for an acknowledge from the Kafka broker. Basically, it means there's no guarantee that messages have been received completely by the broker.
  - 1 means the appender waits for the acknowledge only from the leader. If the leader falls down, its possible messages are lost if the replicas have not yet been created on the followers.
  - all means the appender waits for the acknowledge from the leader and all followers. This mode is the most reliable as the appender will receive the acknowledge only when all replicas have been created. NB: this mode doesn't make sense if you have a single node Kafka broker or a replication factor set to 1.
- the retries defines the number of retries performed by the appender in case of error. The default value is 0 meaning no retry at all.
- the batch.size defines the size of the batch records. The appender will attempt to batch records together into fewer requests whenever multiple records are being sent to the same Kafka partition. The default value is 16384.
- the buffer.memory defines the size of the buffer the appender uses to send to the Kafka broker. The default value is 33554432.
- the key.serializer defines the fully qualified class name of the Serializer used to serialize the keys. The default is a String serializer (org.apache.kafka.common.serialization.StringSerializer).
- the value.serializer defines the full qualified class name of the Serializer used to serialize the values. The default is a String serializer (org.apache.kafka.common.serialization.StringSerializer).
- the request.timeout.ms is the time the producer wait before considering the message production on the broker fails (default is 5s).
- the max.request.size is the max size of the request sent to the broker (default is 2097152 bytes).
- the topic defines the name of the topic where to send data on the Kafka broker.

It's also possible to enable SSL security (with Kafka 0.9.x) using the SSL properties.

1.3.8. REDIS

The Decanter Redis appender sends the data (collected by the collectors) to a Redis broker.

The decanter-appender-redis feature installs the Redis appender:

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

This feature also installs a default etc/org.apache.karaf.decanter.appender.redis.cfg configuration file containing:

```
########################################
# Decanter Redis Appender Configuration
########################################

#
# Location of the Redis broker
# It's possible to use a list of brokers, for instance:
# host= locahost:6389,localhost:6332,localhost:6419
#
# Default is localhost:6379
#
address=localhost:6379

#
# Define the connection mode.
# Possible modes: Single (default), Master_Slave, Sentinel, Cluster
#
mode=Single

#
# Name of the Redis map
# Default is Decanter
#
map=Decanter

#
# For Master_Slave mode, we define the location of the master
# Default is localhost:6379
#
#masterAddress=localhost:6379

#
# For Sentinel model, define the name of the master
# Default is myMaster
#
#masterName=myMaster

#
# For Cluster mode, define the scan interval of the nodes in the cluster
# Default value is 2000 (2 seconds).
#
#scanInterval=2000
```

This file allows you to configure the Redis broker to use:

- the address property contains the location of the Redis broker
- the mode property defines the Redis topology to use (Single, Master_Slave, Sentinel, Cluster)
- the map property contains the name of the Redis map to use
- the masterAddress is the location of the master when using the Master_Slave topology
- the masterName is the name of the master when using the Sentinel topology
- the scanInternal is the scan interval of the nodes when using the Cluster topology

1.3.9. MQTT

The Decanter MQTT appender sends the data (collected by the collectors) to a MQTT broker.

The decanter-appender-mqtt feature installs the MQTT appender:

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

This feature installs a default etc/org.apache.karaf.decanter.appender.mqtt.cfg configuration file containing:

```
#server=tcp://localhost:9300
#clientId=decanter
#topic=decanter
```

This file allows you to configure the location and where to send in the MQTT broker:

- the server contains the location of the MQTT broker
- the clientId identifies the appender on the MQTT broker
- the topic is the name of the topic where to send the messages

1.3.10. CASSANDRA

The Decanter Cassandra appender allows you to store the data (coming from the collectors) into an Apache Cassandra database.

The decanter-appender-cassandra feature installs this appender:

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

This feature installs the appender and a default etc/org.apache.karaf.decanter.appender.cassandra.cfg configuration file containing:

```
########################################
# Decanter Cassandra Appender Configuration
########################################

# Name of Keyspace
keyspace.name=decanter

# Name of table to write to
table.name=decanter

# Cassandra host name
cassandra.host=

# Cassandra port
cassandra.port=9042
```

- the keyspace.name property identifies the keyspace used for Decanter data
- the table.name property defines the name of the table where to store the data
- the cassandra.host property contains the hostname or IP address where the Cassandra instance is running (default is localhost)
- the cassandra.port property contains the port number of the Cassandra instance (default is 9042)

1.3.11. INFLUXDB

The Decanter InfluxDB appender allows you to store the data (coming from the collectors) as a time series into a InfluxDB database.

The decanter-appender-influxdb feature installs this appender:

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

This feature installs the appender and a default etc/org.apache.karaf.decanter.appender.influxdb.cfg configuration file containing:

```
########################################
# Decanter InfluxDB Appender Configuration
########################################

# URL of the InfluxDB database
url=

# InfluxDB server username
#username=

# InfluxDB server password
#password=

# InfluxDB database name
database=decanter
```

- url property is mandatory and define the location of the InfluxDB server
- database property contains the name of the InfluxDB database. Default is decanter.
- username and password are optional and define the authentication to the InfluxDB server.

1.3.12. MONGODB

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

The decanter-appender-mongodb feature installs this appender:

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

This feature installs the appender and a default etc/org.apache.karaf.decanter.appender.mongodb.cfg configuration file containing:

```
################################
# Decanter MongoDB Configuration
################################

# MongoDB connection URI
#uri=mongodb://localhost

# MongoDB database name
#database=decanter

# MongoDB collection name
#collection=decanter
```

- the uri property contains the location of the MongoDB instance
- the database property contains the name of the MongoDB database

- the collection property contains the name of the MongoDB collection

1.3.13. NETWORK SOCKET

The Decanter network socket appender sends the collected data to a remote Decanter network socket collector.

The use case could be to dedicate a Karaf instance as a central monitoring platform, receiving collected data from the other nodes.

The decanter-appender-socket feature installs this appender:

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

This feature installs the appender and a default etc/org.apache.karaf.decanter.appender.socket.cfg configuration file containing:

```
# Decanter Socket Appender

# Hostname (or IP address) where to send the collected data
#host=localhost

# Port number where to send the collected data
#port=34343
```

- the host property contains the hostname or IP address of the remote network socket collector
- the port property contains the port number of the remote network socket collector

1.3.14. ORIENTDB

The Decanter OrientDB appender stores the collected data into OrientDB Document database.

You can use an external OrientDB instance or you can use an embedded instance provided by Decanter.

ORIENTDB APPENDER

The decanter-appender-orientdb feature installs the OrientDB appender.

This feature installs the etc/org.apache.karaf.decanter.appender.orientdb.cfg configuration file allowing you to setup the location of the OrientDB database to use:

```
################################
# Decanter OrientDB Configuration
################################

# OrientDB connection URL
#url=remote:localhost/decanter

# OrientDB database username
#username=root

# OrientDB database password
#password=decanter
```

where:

- url is the location of the OrientDB Document database. By default, it uses remote:localhost/decanter corresponding to the OrientDB embedded instance.
- username is the username to connect to the remote OrientDB Document database.
- password is the password to connect to the remote OrientDB Document database.

ORIENTDB EMBEDDED INSTANCE

For production, we recommend to use a dedicated OrientDB instance. The following feature is not recommended for production.

The orientdb feature starts an OrientDB embedded datase. It also installs the etc/orientdb-server-config.xml configuration file allowing you to configure the OrientBD instance:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<orient-server>
  <handlers>
    <handler class="com.orientechnologies.orient.graph.handler.OGraphServerHandler">
      <parameters>
        <parameter value="true" name="enabled"/>
        <parameter value="50" name="graph.pool.max"/>
      </parameters>
    </handler>
    <handler class="com.orientechnologies.orient.server.handler.OJMXPlugin">
      <parameters>
        <parameter value="false" name="enabled"/>
        <parameter value="true" name="profilerManaged"/>
      </parameters>
    </handler>
    <handler class="com.orientechnologies.orient.server.handler.OServerSideScriptInterpreter">
      <parameters>
        <parameter value="true" name="enabled"/>
        <parameter value="SQL" name="allowedLanguages"/>
      </parameters>
    </handler>
  </handlers>
```

```
    <network>
      <protocols>
        <protocol
implementation="com.orientechnologies.orient.server.network.protocol.binary.ONetworkProtocolB
inary" name="binary"/>
        <protocol
implementation="com.orientechnologies.orient.server.network.protocol.http.ONetworkProtocolHtt
pDb" name="http"/>
      </protocols>
      <listeners>
        <listener protocol="binary" socket="default" port-range="2424-2430" ip-address="0.0.0.0"/>
        <listener protocol="http" socket="default" port-range="2480-2490" ip-address="0.0.0.0">
          <commands>
            <command
implementation="com.orientechnologies.orient.server.network.protocol.http.command.get.OServer
CommandGetStaticContent" pattern="GET|www GET|studio/ GET| GET|*.htm GET|*.html
GET|*.xml GET|*.jpeg GET|*.jpg GET|*.png GET|*.gif GET|*.js GET|*.css GET|*.swf GET|*.ico
GET|*.txt GET|*.otf GET|*.pjs GET|*.svg GET|*.json GET|*.woff GET|*.woff2 GET|*.ttf GET|*.svgz"
stateful="false">
              <parameters>
                <entry value="Cache-Control: no-cache, no-store, max-age=0, must-
revalidate\r\nPragma: no-cache" name="http.cache:*.htm *.html"/>
                <entry value="Cache-Control: max-age=120" name="http.cache:default"/>
              </parameters>
            </command>
            <command
implementation="com.orientechnologies.orient.graph.server.command.OServerCommandGetGephi
" pattern="GET|gephi/*" stateful="false"/>
          </commands>
          <parameters>
            <parameter value="utf-8" name="network.http.charset"/>
            <parameter value="true" name="network.http.jsonResponseError"/>
          </parameters>
        </listener>
      </listeners>
    </network>
    <storages/>
    <users>
    </users>
    <properties>
      <entry value="1" name="db.pool.min"/>
      <entry value="50" name="db.pool.max"/>
      <entry value="false" name="profiler.enabled"/>
    </properties>
    <isAfterFirstTime>true</isAfterFirstTime>
</orient-server>
```

Most of the values can be let as they are, however, you can tweak some:

- <listener/> allows you to configure the protocol and port numbers used by the OrientDB instance. You can define the IP address on which the instance is bound (ip-address), the port numbers range to use (port-range) for each protocol (binary or http).
- the db.pool.min and db.pool.max can be increased if you have a large number of connections on the instance.

1.3.15. DROPWIZARD METRICS

The Dropwizard Metrics appender receives the harvested data from the dispatcher and pushes to a Dropwizard Metrics MetricRegistry. You can register this MetricRegistry in your own application or use a Dropwizard Metrics Reporter to "push" these metrics to some backend.

The decanter-appender-dropwizard feature provides the Decanter event handler registering the harvested data ino the MetricRegistry:

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

1.3.16. TIMESCALEDB

The Decanter TimescaleDB appender stores the collected data into TimescaleDB database.

You have to install a TimescaleDB before using the appender.

You can install a test database with Docker for dev:

```
docker run -d --name timescaledb -p 5432:5432 -e POSTGRES_PASSWORD=decanter -e
POSTGRES_USER=decanter -e POSTGRES_DATABASE=decanter timescale/timescaledb
```

TIMESCALEDB APPENDER

The decanter-appender-timescaledb feature installs the TimescaleDB appender.

As TimescaleDB is a PostgreSQL database extension, the **timescaledb** feature will install all required features to configure your datasource (jdbc, jndi, postgreSQL driver, pool datasource).

This feature installs the etc/org.apache.karaf.decanter.appender.timescaledb.cfg configuration file allowing you to setup the location of the TimescaleDB database to use:

```
################################
# Decanter TimescaleDB Configuration
################################

# DataSource to use
dataSource.target=(osgi.jndi.service.name=jdbc/decanter-timescaledb)

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

# Marshaller to use (json is recommended)
marshaller.target=(dataFormat=json)
```

where:

- datasource.target 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).
- table.name property contains the table name in the database. The Decanter JDBC appender automatically activates the Timescale extenssion, creates the table for you and migrates the table to a TimescaleDB hypertable. The table is simple and contains just two column:
  - timestamp as BIGINT
  - content as TEXT
- marshaller.target is the marshaller used to serialize data into the table.

WEBSOCKET SERVLET

The decanter-appender-websocket-servlet feature exposes a websocket on which clients can register. Then, Decanter will send the collected data to the connected clients.

It’s very easy to use. First install the feature:

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

The feature registers the WebSocket endpoint on http://localhost:8181/decanter-websocket by default:

```
karaf@root()> http:list
ID │ Servlet                 │ Servlet-Name   │ State       │ Alias               │ Url
──────────────────────────────────────────────────────────────────────────────────────────
55 │ DecanterWebSocketServlet │ ServletModel-2 │ Deployed    │ /decanter-websocket │ [/decanter-websocket/*]
```

The alias can be configured via the etc/org.apache.karaf.decanter.appender.websocket.servlet.cfg configuration file installed by the feature.

You can now register your websocket client on this URL. You can use curl as client to test:

```
curl --include \
    --no-buffer \
    --header "Connection: Upgrade" \
    --header "Upgrade: websocket" \
    --header "Host: localhost:8181" \
    --header "Origin: http://localhost:8181/decanter-websocket" \
    --header "Sec-WebSocket-Key: SGVsbG8sIHdvcmxkIQ==" \
    --header "Sec-WebSocket-Version: 13" \
    http://localhost:8181/decanter-websocket
```

PROMETHEUS

The decanter-appender-prometheus feature collects and exposes metrics on prometheus:

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

The feature registers the Prometheus HTTP servlet on http://localhost:8181/decanter/prometheus by default:

```
karaf@root()> http:list
ID │ Servlet        │ Servlet-Name   │ State       │ Alias               │ Url
───────────────────────────────────────────────────────────────────────────────────────
51 │ MetricsServlet │ ServletModel-2 │ Deployed    │ /decanter/prometheus │
[/decanter/prometheus/*]
```

You can change the servlet alias in etc/org.apache.karaf.decanter.appender.prometheus.cfg configuration file:

```
################################################################################
#
#    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.
#
################################################################################

#################################################
# Decanter Prometheus Appender Configuration
#################################################

# Prometheus HTTP servlet alias
#alias=/decanter/prometheus
```

The Decanter Prometheus appender exports io.prometheus* packages, meaning that you can simple add your metrics to the Decanter Prometheus servlet. You just have to import io.prometheus* packages and simple use the regular Prometheus code:

```
class YourClass {
  static final Gauge inprogressRequests = Gauge.build()
     .name("inprogress_requests").help("Inprogress requests.").register();

  void processRequest() {
    inprogressRequests.inc();
    // Your code here.
    inprogressRequests.dec();
  }
}
```

Don't forget to import io.prometheus* packages in your bundle MANIFEST.MF:

```
Import-Package: io.prometheus.client;version="[0.8,1)"
```

That's the only thing you need: your metrics will be available on the Decanter Prometheus servlet (again on http://localhost:8181/decanter/prometheus by default).

1.4. ALERTING

Decanter provides an alerting feature. It allows you to check values in harvested data (coming from the collectors) and send alerts when the data is not in the expected state.

1.4.1. SERVICE

The alerting service is the core of Decanter alerting.

It's configured in etc/org.apache.karaf.decanter.alerting.service.cfg configuration file where you define the alert rules:

```
rule.my="{'condition':'message:*','level':'ERROR'}
```

The rule name has to start with rule. prefix (see rule.my here).

The rule definition is in JSON with the following syntax:

```
{
   'condition':'QUERY',
   'level':'LEVEL',
   'period':'PERIOD',
   'recoverable':true|false
}
```

where:

- condition is a Apache Lucene query (https://lucene.apache.org/core/8_5_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#package.description). For instance:
  - message:foo* selects all events with message containing any string starting with foo
  - message:* AND other:* selects all events with message and other containing anything
  - threadCount:[200 TO *] selects all events with threadCount greater than 200
  - counter:[20 TO 100) selects all events with counter between 20 and 100 (included)
  - foo:bar OR foo:bla selects all events with foo containing bar or bla
- level is a string where you can set whatever you want to define the alert level. By default, it's WARN.
- period is optional and allows you to define a validity period for a condition. It means that the condition should match for the period duration and, if so, the alert with be thrown after the period. The period is a string like this:
  - 5SECONDS
  - 10MINUTES
  - 2HOURS
- recoverable is a flag that define if the alert can be recover or not. By default it's false. The main difference is the number of alert events you will have. If not recoverable, you will have an alert for each event matching the condition. If recoverable, you will have a single alert the first time an event matches the condition, and another alert (back to normal) when the alert is back (not matching the condition).

You can use any event property in the rule condition. The alert service automatically add:

- alertUUID is a unique string generated by the alert service
- alertTimestamp is the alert timestamp added by the alert service

1.4.2. ALERTERS

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

Apache Karaf Decanter provides ready to use alerters.

LOG

The Decanter Log alerter logs a message for each alert.

The decanter-alerting-log feature installs the log alerter:

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

This alerter doesn't need any configuration.

E-MAIL

The Decanter e-mail alerter sends an e-mail for alerts.

The decanter-alerting-email feature installs the e-mail alerter:

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

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

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

# From e-mail address
from=

# To e-mail addresses, you can define here a list of recipient separated with comma
# For example: to=mail1@example.org,mail2@example.org,mail3@example.org
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=

# e-mail velocity templates
#subject.template=/path/to/subjectTemplate.vm
#body.template=/path/to/bodyTemplate.vm
#body.type=text/plain
```

- 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
- the subject.template property allows you to provide your own Velocity (http://velocity.apache.org) template to create the subject of the message
- the body.template property allows you to provide your own Velocity (http://velocity.apache.org) template to create and format the body of the message
- the body.type property allows you to define the message content type, depending if you send HTML or plain text message.

Optionally, you can add:

- cc to add email carbon copy
- bcc to add email blind carbon copy

The email alerter is also able to use collected data properties.

For instance, subject can look like This is my ${property} where ${property} is replaced by the property value.

The email alerter is also able to use collected data properties for subject or body (including replacement). It looks for body.template.location and subject.template.location collected data properties.

For instance, a body Velocity template looks like this:

```
#if ($event.get("alertBackToNormal") == true)
$event.get("alertLevel") alert: $event.get("alertAttribute") was out of the pattern
$event.get("alertPattern") but back to normal now
#else
$event.get("alertLevel") alert: $event.get("alertAttribute") is out of the pattern
$event.get("alertPattern")
#end

Details:
#foreach ($key in $event.keySet())
 $key : $event.get($key)
#end
```

where $event is the map containing all event properties.

CAMEL

The Decanter 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-alerting-camel feature installs the Camel alerter:

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

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

```
#
# Decanter Camel alerter
#

# alert.destination.uri defines the Camel endpoint URI where
# Decanter send the 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>
```

USING EXISTING APPENDERS

Actually, a Decanter alerter is a "regular" Decanter appender. The different is the events topic listening by the appender. By default, the appenders listen on decanter/collect/*. To "turn" an appender as an alerter, it just has to listen on decanter/alert/*.

For instance, you can create a new instance of elasticsearch appender by creating etc/org.apache.karaf.decanter.appender.elasticsearch-myalert.cfg containing:

```
event.topics=decanter/alert/*
...
```

With this configuration, you have a elasticearch alerter that will store the alerts into a elasticsearch instance.

1.5. PROCESSORS

Decanter Processors are optional. They receive data from the collectors, apply a processing logic on the received event, and send a new event to the appenders.

The processors are listening for incoming events on decanter/collect/* dispatcher topics and send processed events to decanter/process/* dispatcher topics. By default, the appenders are listening on decanter/collect/* topics. If you want to append processed events, you have to configure the appenders to listen on decanter/process/* topics. To do that, you just have to change appender configuration with:

```
event.topics=decanter/process/*
```

It's possible to "chain" processors thanks to the topics. For instance, you can have the first processor listening on decanter/collect/* topic (containing events coming from the collectors), and sending processed events to decanter/process/first. Then, a second processor can listen on decanter/process/first topic and send processed data to decanter/process/second. Finally, at the end of the chain, you have to configure the appenders to listen on decanter/process/second.

1.5.1. PASS THROUGH

This processor doesn't implement any concrete logic. It's for the example how to implement a processor.

You can install this processor using the decanter-processor-passthrough feature:

```
karaf@root()> feature:install decanter-processor-passthrough
```

1.5.2. AGGREGATE

This processor "merges" several incoming events in a single one that is sent periodically.

You can install this processor using the decanter-processor-aggregate feature:

```
karaf@root()> feature:install decanter-processor-aggregate
```

By default, the "merged" event is sent every minute. You can change this using the period configuration.

You can provisiong etc/org.apache.karaf.decanter.processor.aggregate.cfg configuration file with:

```
period=120 # this is the period in seconds
target.topics=decanter/process/aggregate # that's the default target topic
```

You can also decide if a known property is overwritten in the aggregator or appended.

By default, properties are not overwritten, meaning that it's prefixed by the event index in the aggregator:

```
0.foo=first
0.other=bar
1.foo=second
1.other=bar
```

In the processor etc/org.apache.karaf.decanter.processor.aggregate.cfg configuration file, you can enable overwrite:

```
overwrite=true
```

Then, if a property already exist in the aggregator, its value will be overwritten by the new event value received in the aggregator.

2. DEVELOPER GUIDE

2.1. 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 checker to send the alerts. The 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.

2.2. 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.

2.2.1. 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 implementing 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 dispatcher;

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

    @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);
      dispatcher.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 dispatcher = (EventAdmin) tracker.waitForService(10000);
        collector = new BundleCollector(dispatcher);
    }

    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.

2.2.2. 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 dispatcher;

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

  @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);
    dispatcher.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 dispatcher = tracker.waitForService(10000);
     HelloCollector collector = new HelloCollector(dispatcher);

     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 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.

2.3. CUSTOM APPENDER

A Decanter Appender is an OSGi EventAdmin EventHandler: it listens to 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 registers 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>1.1.0-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).

2.4. CUSTOM ALERTER

A Decanter Alerter is basically a special kind of appender.

It's an OSGi EventAdmin EventHandler: it listens to decanter/alert/* EventAdmin topics, and receives the alerting data coming from the 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 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).

2.5. CUSTOM PROCESSOR

A Decanter Processor is listening OSGi EventAdmin events from decanter/collect/* and send a new OSGi EventAdmin event to decanter/process/* `

To see how to implement your own processor, you can take a look on the pass through processor: https://github.com/apache/karaf-decanter/blob/master/processor/passthrough/src/main/java/org/apache/karaf/decanter/processor/passthrough/PassThroughProcessor.java