KIP-2 Metrics

Status

Current state: *Complete*
JIRA: https://issues.apache.org/jira/browse/KNOX-643

Motivation

The JIRA and discussions have the various requirements captured for the ability to gather metrics of during the gateway's processing pipeline and request /response flow. The initial attempt to satisfy some of the requirements is to provide a very simple abstraction to hide the details of the dropwizard metrics library and expose some of the basic and most requested metrics. As a side note, all the requirements in KNOX-643 are not going to be captured by this KIP and the associated work. The following requirements are going to be addressed:

1. Ability to get time taken for request/reponse coming from the client and the frequency at various time intervals at the service level
2. Ability the get the time and frequency of request/responses to the backend service component.
3. The number of open connections to the backend service component.
4. API to add/extend the metrics capabilities.
5. Ability to report the metrics to reporting engines like Graphite and Ambari Metrics Service.

Design

The dropwizard metrics library was selected after some comparative analysis of similar libraries that allow for instrumenting code so that metrics can be gathered at runtime. This document is leaving out that analysis and asserting the result that the dropwizard metrics library was essentially the easiest to use API that provided the functionality we were looking for, had the appropriate licensing and was most frequently used in other Apache projects.
The desire however as always is to provide a layer of abstraction, leaving the possibility open to future changes or adoption of other libraries. The pattern used in the API design is of being able to provide instrumented versions of a class or an interface and not yet exposing the detailed measuring instruments like Guages, meters etc.
The MetricsService API therefore looks like this:

```
public interface MetricsService extends Service {

  <T> T getInstrumented(T instanceClass);

  <T> T getInstrumented(Class<T> clazz);

}
```

The MetricsService is implemented as a Gateway Service, the details of which can be found in the dev guide. It is therefore accessible to all Topology deployments so that per topology metrics can be done and of course aggregation can be done as well at the gateway level.

Plugging in a new Instrumented Class

The standard ServiceLoader mechanism is used to find and load classes that can provide instrumented classes. The two interfaces involved are :

```
public interface InstrumentationProvider<T> {

  T getInstrumented(MetricsContext metricsContext);

 T getInstrumented(T instanceClass, MetricsContext metricsContext);

}
```

and

```
public interface InstrumentationProviderDescriptor {

  Map<Class<?>, InstrumentationProvider> providesInstrumentation();
}
```

The service loader mechanism will look up InstrumentationProviderDescriptor classes so a provider-configuration file needs to be provided for this implementation in a resource directory META-INF/services.

Reporting

```
public interface MetricsReporter {

  String getName();

  void init(GatewayConfig config) throws MetricsReporterException;

  void start(MetricsContext metricsContext) throws MetricsReporterException;

  void stop() throws MetricsReporterException;

  boolean isEnabled();
}
```

Plugging in a new Reporter

The ServiceLoader pattern is used again here so all you need to do to add a new reporter is to implement the MetricsReporter interface and provide a provider-configuration file in a resource directory META-INF/services.

Since the GatewayConfig is passed to the reporter instance on init, any configuration that needs to be provided to the reporter needs to be wired up through the GatewayConfig interface. This essentially means that configuration can be provided in the gateway-site.xml file.

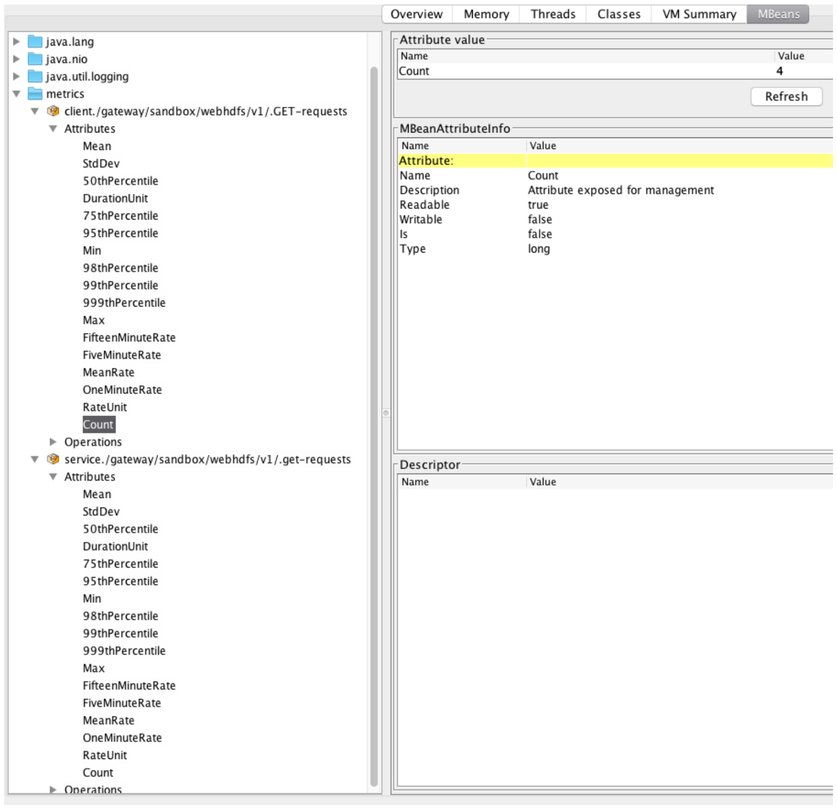
Initial metrics and naming

The initial instrumentation implementation leverages dropmetrics API as well as some built-in functionality for gathering metrics in the gateway's Dispatch code as well as incoming requests from the client. The Dispatch code uses Apache's httpclient library so if metrics is enabled we can switch out the regular httpclient for an instrumented one. Thus gathering timing, count and connection information for all our dispatch requests.

Similarly the GatewayFilter class has an instrumented version to collect data on incoming requests from the clients.

The naming of these metrics has to be a bit elaborate to allow for enough namespacing. The client requests coming into the gateway all start with 'client' and the dispatch requests to a backing service all start with 'service'. After that it is the request URI followed by the request method (GET, PUT, POST etc).

Below is a screen shot of some of the attributes and names provided as seen in a JMX viewer.



```
▶ java.lang
▶ java.nio
▶ java.util.logging
▼ metrics
  ▼ client./gateway/sandbox/webhdfs/v1/.GET-requests
    ▼ Attributes
        Mean
        StdDev
        50thPercentile
        DurationUnit
        75thPercentile
        95thPercentile
        Min
        98thPercentile
        99thPercentile
        999thPercentile
        Max
        FifteenMinuteRate
        FiveMinuteRate
        MeanRate
        OneMinuteRate
        RateUnit
        Count
    ▶ Operations
  ▼ service./gateway/sandbox/webhdfs/v1/.get-requests
    ▼ Attributes
        Mean
        StdDev
        50thPercentile
        DurationUnit
        75thPercentile
        95thPercentile
        Min
        98thPercentile
        99thPercentile
        999thPercentile
        Max
        FifteenMinuteRate
        FiveMinuteRate
        MeanRate
        OneMinuteRate
        RateUnit
        Count
    ▶ Operations
```

Attribute value

Name	Value
Count	4

MBeanAttributeInfo

Name	Value
Attribute:	
Name	Count
Description	Attribute exposed for management
Readable	true
Writable	false
Is	false
Type	long

Descriptor

Name	Value

In the example above a curl request like this:

curl -iku guest:guest-password -X GET 'https://localhost:8443/gateway/sandbox/webhdfs/v1/?op=GETHOMEDIRECTORY'

results in two mbeans:

1. client./gateway/sandbox/webhdfs/v1/.GET-requests
2. service./gateway/sandbox/webhdfs/v1/.GET-requests

The attributes provided are:

Mean

StdDev

DurationUnit

50thPercentile

75thPercentile

98thPercentile

99thPercentile

999thPercentile

Min

Max

FifteenMinuteRate

FiveMinuteRate

OneMinuteRate

MeanRate

RateUnit

Count

Graphite+Grafana

Config

The initial configuration is mainly for turning on or off the metrics collection and then enabling reporters with their required config. The two initial reporters implemented are JMX and Graphite.

gateway.metrics.enabled

```
Turns on or off the metrics, default is 'true'
```

gateway.jmx.metrics.reporting.enabled

```
Turns on or off the jmx reporter, default is 'true'
```

gateway.graphite.metrics.reporting.enabled

```
Turns on or off the graphite reporter, default is 'false'
```

```
gateway.graphite.metrics.reporting.host
gateway.graphite.metrics.reporting.port
gateway.graphite.metrics.reporting.frequency
```

The above are the host, port and frequency of reporting (in seconds) parameters for the graphite reporter.

Future work

From the list of requirements one of the glaring holes is that of getting more metrics out of Knox’s Shiro/LDAP provider. I believe this requirement comes from more of a debugging mindset when faced with issues in the field, but may have a broader appeal so need some validation.
The other main task which is possibly a near future item is to provide for additional reporters. Specifically of interest would be a reporter that sends data to the Ambari Metrics Service. This would provide a convenient solution for viewing the metrics when in a hadoop deployment that has Ambari available.