MetricsConfiguration

This is a somewhat experimental feature, and many of the elements should be considered preliminary at this stage. In particular, the names of individual metrics are **not** part of the public API of the software at this time and are subject to change as we gain additional experience and feedback. You should take this into account if you build any tools for collection or monitoring and allow for changes to the specific names used.

Current File(s): *conf/admin/metrics.xml*, *conf/admin/general-admin.xml*, *conf/logback.xml*

Format: Native Spring, Logback

Overview

V3.3 introduces a new framework for instrumentation, diagnostics, and performance management that complements the logging support available previously and integrates with it to allow tuning of instrumentation overhead. The framework is based on the Metrics library and includes a number of related features:

- some built-in metrics providing low-level system information, software versions, etc.
- a small number of built-in metrics exposed by the IdP itself
- a facillity for installing timers and counters to measure request handling
- an API for accessing metrics (all or a subset) in JSON format
- the ability to schedule reports of metrics in JSON format out to an http(s) collection point

Most of these features are configured whole or in part via *conf/admin/metrics.xml* and are discussed below.

General Configuration

The core bean in the metric system is a MetricRegistry that stores all of the active metrics in the system for use by various other reporting objects. Certain kinds of metrics are subject to your control and can be selectively installed or skipped based on what gets registered, at startup. By default various sets of metrics are installed for you via a bean in *metrics.xml* that inherits from **shibboleth.metrics.RegisterMetricSets,** which is a way of using Spring to call the registration method. The pre-installed metrics largely overlap with the information available from the now-deprecated status page.

Also provided by default, but commented out, are references to a number of lower level metric sets related to the JVM.

The way to think about these particular metrics is that they're "external" to the IdP for the most part, in the sense that they're objects that access public interfaces of IdP, OpenSAML, and Java components, and use them to report information. Because they're external, they can be selectively installed (or not) just by registering them (or not).

A lot of the other interesting metrics in the system are "internal", in the sense that they're implemented either wholly, or in conjunction with, internal classes themselves, and typically are not something you specifically need to register yourself. Instead, you have the ability to enable or disable all metrics using a MetricFilter.

Logger-Driven Filtering

The registry of metrics is extended by Shibboleth with the ability to "filter" metrics by reusing the logging configuration's ability to adjust the logging level of particular categories hierarchically. By naming metrics in a similar way, it's possible to effectively enable and disable metrics at runtime, which can have very small (or in rare cases, less small) impacts on performance. You may want to leave a metric configured all the time, but toggle it on only at particular times. Using the logging layer is easy, flexible, reloadable, and ignorable if you don't care about the overhead.

Every metric has a name in dotted-separator notation (i.e., the same as a logging category) and you can control the on/off status of a particular metric or set of metrics by manipulating the level of a corresponding logging category with the prefix "metrics." That is, a metric named "net.shibboleth.idp.version" is controlled by a logging category named "metrics.net.shibboleth.idp.version".

By default, all metrics are enabled at the "INFO" level.

Reporting

There are two built-in mechanisms provided for reporting out metrics, along with a variety of options available within the Metrics library itself that deployers may choose to wire in with Spring. For example, it's possible to expose metrics via JMX, through logging, via the console, to databases, etc.

The two generic approaches that come "ready to deploy" are both centered around use of JSON as a reporting format, and both pull (via a REST API) and push (via an HTTP reporter) are provided.

REST API

As a (greatly enhanced) replacement for the old status page, an Administrative flow is provided for access to specific metrics, named groups, or all of the metrics in the system, in JSON or JSONP format.

High level configuration of this flow is handled by *conf/admin/general-admin.xml*, but in most cases the defaults suffice and the real configuration, if any, takes place in *conf/admin/metrics.xml*.

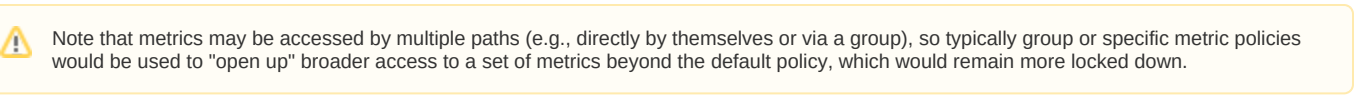
There are two main features you can configure:

- Naming custom groupings of metrics for ease of access, and access control.
- Applying specific access control policies to metrics or groups of metrics.

By default, the REST API provides access to either all metrics (via */idp/profile/admin/metrics*), or to a specific metric by attaching the name to the end of the path (e.g., */idp/profile/admin/metrics/net.shibboleth.idp.version).*

In addition, a number of predefined "groups" are configured for you by default, in a bean named **shibboleth.metrics.MetricGroups**. The groups associate an object that implements the MetricFilter interface with a shorthand name, and allow you to access all the metrics that satisfy the filter by using the group name on the end of the path in place of a specific metric name. For example, the default configuration allows a number of metrics related to the metadata layer to be accessed via */idp/profile/admin/metrics/metadata*.

The other purpose of these groups is for access control. By default, access to the entire REST API is governed by a default policy identified by **shibboleth.metrics.DefaultAccessPolicy**. If you need more fine-grained control, you can fill in the map defined by **shibboleth.metrics.AccessPolicyMap** with entries connecting a metric name or metric group name with a named policy to apply.



A couple of additional obscure features: it's possible to control the "Access-Control-Allow-Origin" header and turn the feed into a JSONP callback using String beans (see the Reference below) for more exotic uses of the API.

HTTP Reporters

As an alternative to a pull model, the IdP can push metric information out to an HTTP server as a collection point. This is not enabled by default, but some basic configuration support is provided inside a comment in *conf/admin/metrics.xml*.

You can define any number of reporters using the parent bean **shibboleth.metrics.HTTPReporter**. The reporter bean is configured with the URL to access, and can be injected with an HttpClient bean for control over many aspects of the HTTP connection, such as timeouts, how to evaluate a server's TLS certificate, etc. It's also possible to add a MetricFilter constructor argument to limit which metrics are actually included.

In addition to defining the bean itself, it must be "scheduled" by calling its "start" method with a timer interval, as illustrated in the default file.

The format of the data is identical to the format returned by the REST API.

Per-Profile Instrumentation

In addition to the built-in metrics, support exists across a number of components (primarily the Action beans and a few of the major services like the Attribute Resolver and Attribute Filter) to add counters and timers to specific requests. This is done by providing a script (or alternative Java code, but a script is simpler) to add the counters and timers to install.

The names of the metrics are arbitrary; the metrics are attached to specific objects based on an identifier for the object that usually will consist of either the Spring bean ID or the simple name of a Java class. This usually requires some specific knowledge of the internals of the system, but advice is available on the support list about how to measure particular parts of the system.

The two kinds of metrics supported are simple counters that measure how often a component runs, and timers that measure the time elapsed between a starting and stopping point. With respect to profile actions, the system will check to see if the execution of the action should be counted and if a timer should be started (at the beginning of execution) or stopped (at the end). The latter means that it's possible to start a timer in one action and stop it in another, bracketing larger parts of a request than just a single action.

(The above is best explained with an example, so keep reading.)

Control over the installation of these metrics is managed by a bean of type Function<ProfileRequestContext,Boolean> named **shibboleth.metrics.MetricStrategy**. The function's job is to manipulate a MetricContext child context to configure the metrics to activate for the request. The javadoc for that child context class describes the methods available to add counters and timers.

The example provided by default illustrates a simple use case: measuring the time it takes to resolve attributes during the request. In the example, a timer named "idp.attribute.resolution" is installed that starts when the "ResolveAttributes" action starts and stops when the "FilterAttributes" action finishes. This demonstrates that timers can extend beyond a single step in a flow. You could just as easily time only the "ResolveAttributes" action alone by using the same ID for both parameters.

Timing attribute resolution

```
        <bean id="shibboleth.metrics.MetricStrategy" parent="shibboleth.ContextFunctions.Scripted"
                        factory-method="inlineScript">
                <constructor-arg>
                        <value>
                        <![CDATA[
                                var metricCtx = input.getSubcontext("org.opensaml.profile.context.
MetricContext");
                                metricCtx.addTimer("idp.attribute.resolution",
                                        "ResolveAttributes",
                                        "FilterAttributes"
                                        );
                                true; // Signals success.
                        ]]>
                        </value>
                </constructor-arg>
        </bean>
```

Another example demonstrates the use of a counter tracking the number of times the SAML 1.1 Attribute Query flow executes based on counting each time the "DecodeMessage" action runs. It shows how, because the function is run dynamically, it's possible to conditionally enable metrics based on the specific profile flow being run.

Counter of SAML 1 queries

```
        <bean id="shibboleth.metrics.MetricStrategy" parent="shibboleth.ContextFunctions.Scripted"
                        factory-method="inlineScript">
                <constructor-arg>
                        <value>
                        <![CDATA[
                                var profileType = Java.type("net.shibboleth.idp.saml.saml1.profile.config.
AttributeQueryProfileConfiguration");
                                if (profileType.PROFILE_ID.equals(input.getProfileId())) {
                                        metricCtx = input.getSubcontext("org.opensaml.profile.context.
MetricContext");
                                        metricCtx.addCounter("idp.profile.saml1.attributeQueries",
"DecodeMessage");
                                }
                                true; // Signals success.
                        ]]>
                        </value>
                </constructor-arg>
        </bean>
```

Finally, here's a way to take advantage of how early the strategy function runs to capture (approximately) the time at the start of a request so that it can be included in the audit log, allowing it to be used with other audit fields to compute the wall clock time to process requests.

Capture start time of requests in an audit field

```
        <bean id="shibboleth.metrics.MetricStrategy" parent="shibboleth.ContextFunctions.Scripted"
                        factory-method="inlineScript">
                <constructor-arg>
                        <value>
                        <![CDATA[
                                var dateTimeType = Java.type("org.joda.time.DateTime");
                var auditCtx = input.getSubcontext("net.shibboleth.idp.profile.context.AuditContext", true);
                auditCtx.getFieldValues("ST").add(new dateTimeType().toString("YYYY-MM-dd'T'HH:mm:ss.SSSZZ"));

                true; // Signals success.
                        ]]>
                        </value>
                </constructor-arg>
        </bean>
```

Reference

Beans

Name	Type	Function
shibboleth.metrics.MetricRegistry	FilteredMetricRegistry	Registry of all metrics known to the system
shibboleth.metrics.RegisterMetricSets	MethodInvokingBean	Spring parent bean for invoking the `registerMultiple` method on the registry
shibboleth.metrics.RegisterMetricSet	MethodInvokingBean	Spring parent bean for invoking the `register` method on the registry
shibboleth.metrics.CoreGaugeSet	MetricSet / MetricFilter	Low-level gauges for OS, Java, and memory information
shibboleth.metrics.IdPGaugeSet	MetricSet / MetricFilter	Basic IdP system information (version, uptime)
shibboleth.metrics.LoggingGaugeSet	MetricSet / MetricFilter	Information about the logging service
shibboleth.metrics.AccessControlGaugeSet	MetricSet / MetricFilter	Information about the access control service
shibboleth.metrics.MetadataGaugeSet	MetricSet / MetricFilter	Information about the metadata resolver service
shibboleth.metrics.RelyingPartyGaugeSet	MetricSet / MetricFilter	Information about the RelyingParty configuration service
shibboleth.metrics.NameIdentifierGaugeSet	MetricSet / MetricFilter	Information about the Name Identifier generation service
shibboleth.metrics.AttributeResolverGaugeSet	MetricSet / MetricFilter	Information about the attribute resolver service
shibboleth.metrics.AttributeFilterGaugeSet	MetricSet / MetricFilter	Information about the attribute filter service
shibboleth.metrics.HTTPReporter	HTTPReporter	A schedulable background reporter that sends a JSON feed of metrics to a URL
shibboleth.metrics.MetricGroups	Map<String,MetricFilter>	Associates metrics matching a supplied filter with a string label that "names" that set of metrics
shibboleth.metrics.MetricLevelMap	Map<String,Level>	Optional mapping of metric names to logging levels to associate with the metric
shibboleth.metrics.DefaultAccessPolicy	String	Name of the access control policy to apply to the metrics API in the absence of a more specific policy
shibboleth.metrics.AccessPolicyMap	Map<String,String>	Maps a named metric group/filter from the "shibboleth.metrics.MetricGroups" bean to a named access control policy to apply when accessing that group via the API
shibboleth.metrics.AccessPolicyStrategy	Function<ProfileRequestContext,String>	A mechanism to determine the access control policy to apply to a request to the metrics API, normally relies on the two previous beans but can be replaced if desired
shibboleth.metrics.AllowedOrigin	String	Optional "Access-Control-Allow-Origin" header value to return within REST API response
shibboleth.metrics.JSONPCallback	String	Optional name of JSONP callback function to pass the REST API response
shibboleth.metrics.MetricStrategy	Function<ProfileRequestContext,Boolean>	A hook to provide a function to execute at the beginning of every request that can programmatically enable timers and counters for objects during the execution of that request

Notes

TBD