Build a Monitoring Extension Using Java

A Java monitoring extension enables the AppDynamics Machine Agent to collect custom metrics, which you define and provide, and to report them to the Controller. This is an alternative to adding monitoring extensions using scripts.

When you capture custom metrics with a monitoring extension, they are supported by the same AppDynamics services that you get for the standard metrics captured with the AppDynamics application and machine agents. These services include automatic baselining, anomaly detection, display in the Metric Browser, availability for display on custom dashboards and availability for use in policies to trigger alerts and other actions.

This topic describes the procedure for creating a monitoring extension in Java.

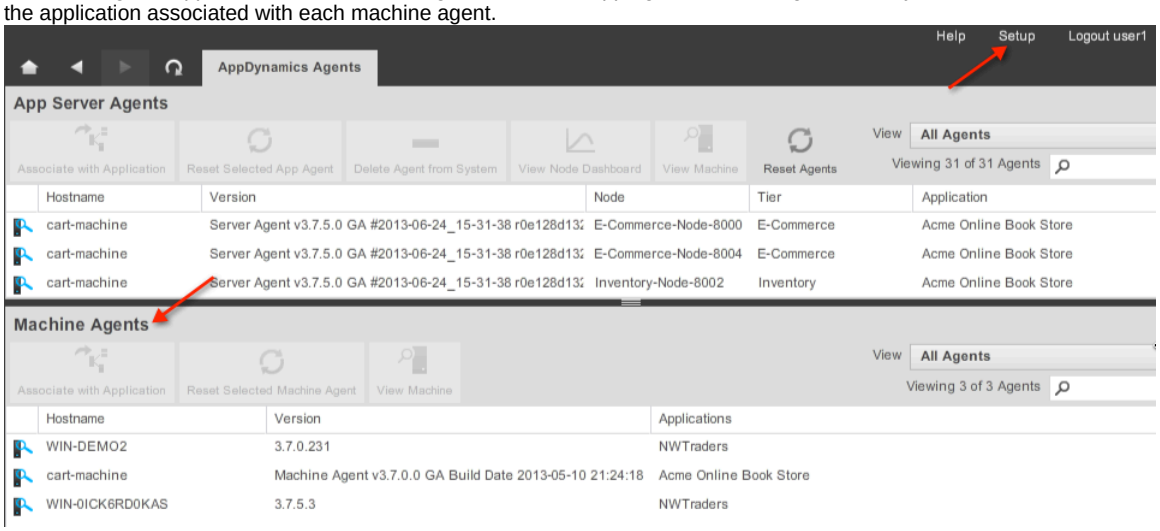
Machine Agent Required

A monitoring extension requires a standalone Machine Agent installed on the machine that hosts the application that you want to monitor.

If you do not know whether you have a Machine Agent installed:

1. In the upper right corner of AppDynamics console, click **Setup**.

2. Click **AppDynamics Agents**.
The list of agents appears with the Machine Agents below the app agents. You can get summary information about the machine and the application associated with each machine agent.



If you do not already have a Machine Agent installed, install one. See Install the Standalone Machine Agent.

To the Machine Agent, a monitoring extension is a task that runs on a fixed schedule and collects metrics. The task can be either

AJavaTask, which executes a task within the machine agent process, or AForkedJavaTask, which executes a task in its own separate process.

Before You Begin

Before creating your own extension from scratch, look at the extensions that have been created and shared among members of the AppDynamics community. The extensions are described and their source is available for free download at

https://github.com/Appdynamics/

New extensions are constantly being added. It is possible that someone has already created exactly what you need or something close enough that you can download it and use it after making a few simple modifications.

Process for Creating a Monitoring Extension in Java

To create a monitoring extension in Java:

1. Create your extension class. See Your Monitoring Extension Class.
2. Create a monitor.xml configuration file. See The monitor.xml File.
3. Create a zip file containing these two files plus any dependent jar files.
4. Create a subdirectory (<your_extension_dir>) in your AppDynamics Machine Agent directory under Machine Agent/monitors.
5. Unzip the zip file into Machine Agent/monitors/<your_extension_dir>.
6. Restart the Machine Agent.

Your Monitoring Extension Class

Create a monitoring extension class by extending the AManagedMonitor class in the com.singularity.ee.agent.systemagent.api package.

You will also need the following helper classes in the same package:

- com.singularity.ee.agent.systemagent.api.MetricWriter:
- com.singularity.ee.agent.systemagent.api.TaskExecutionContext;
- com.singularity.ee.agent.systemagent.api.TaskOutput;
- com.singularity.ee.agent.systemagent.api.exception.TaskExecutionException

The Javadoc for these APIs is available at:

https://rawgithub.com/Appdynamics/java-sdk/master/machine-agent-api/MachineAgentAPIJavadocs/index.html

Your monitor extension class performs these tasks:

- Populates a hash map with the values of the metrics that you want to add to AppDynamics.
  How you obtain these metrics is specific to your environment and to the source from which you derive your custom metrics.
- Describes the metrics using the MetricWriter class.
- Uploads the metrics to the Controller using the execute() method of the TaskOutput class.

Metric Path

All custom metrics processed by the Machine Agent appear in the Application Infrastructure Performance tree in the metric hierarchy. Within the Application Infrastructure Performance tree you specify the metric path, which is the position of the metric in the metric hierarchy, using the "|" character. For example:

- WebServer|
- CustomMetrics|WebServer|
- CustomMetrics|WebServer|XXX|, CustomMetrics|WebServer|YYY|

If the metrics apply to a specific tier, use the colon ":" character to specify the tier name or tier ID. The metric then appears under the specified tier in the metric path. For example:

- Server|Component:ECommerceServer|
- Server|Component:ECommerceServer|CustomMetrics|

Metric Name

Metric names must be unique within the same metric path but need not be unique for the entire metric hierarchy.

It is a good idea to use short metric names so that they will be visible when they are displayed in the Metric Browser.

Prepend the metric path to the metric name when you upload the metrics to the Controller.

Metric Processing

The Controller has various qualifiers for how it processes a metric with regard to aggregation, time rollup and tier rollup. You specify these options with the enumerated types provided by the MetricWriter class. These types are defined below.

Aggregation

The aggregator qualifier specifies how the values reported during a one-minute period are aggregated.

Aggregator Type	Description
METRIC_AGGREGATION_TYPE_AVERAGE	Average of all reported values in the minute. The default operation.
METRIC_AGGREGATION_TYPE_SUM	Sum of all reported values in the minute. This operation causes the metric to behave like a counter.
METRIC_AGGREGATION_TYPE_OBSERVATION	Last reported value in the minute. If no value is reported in that minute, the value from the last time it was reported is used.

Time Roll Up

The time-rollup qualifier specifies how the Controller rolls up the values when it converts from one-minute granularity tables to 10-minute granularity and 60-minute granularity tables over time.

Roll up Strategy	Description
METRIC_TIME_ROLLUP_TYPE_AVERAGE	Average of all one-minute data points when adding it to the 10-minute or 60-minute granularity table.
METRIC_TIME_ROLLUP_TYPE_SUM	Sum of all one-minute data points when adding it to the 10-minute or 60-minute granularity table.
METRIC_TIME_ROLLUP_TYPE_CURRENT	Last reported one-minute data point in that 10-minute or 60-minute interval.

Cluster Roll Up

The cluster-rollup qualifier specifies how the controller aggregates metric values in a tier.

Roll up Strategy	Description
METRIC_CLUSTER_ROLLUP_TYPE_INDIVIDUAL	Aggregates the metric value by averaging the metric values across each node in the tier.
METRIC_CLUSTER_ROLLUP_TYPE_COLLECTIVE	Aggregates the metric value by adding up the metric values for all the nodes in the tier.

For example, if a tier has two nodes, Node A and Node B, and Node A has 3 errors per minute and Node B has 7 errors per minute, the INDIVIDUAL qualifier reports a value of 5 errors per minute and COLLECTIVE qualifier reports 10 errors per minute. INDIVIDUAL is appropriate for metrics such as % CPU Busy where you want the value for each node. COLLECTIVE is appropriate for metrics such as Number of Calls where you want a value for the entire tier.

Sample Monitoring Extension Class

The NGinXMonitor class gets the following metrics from the Nginx Web Server and adds them to the metrics reported by AppDynamics:

- Active Connections: number of active connections
- Accepts: number of accepted requests
- Handled: number of handled requests
- Requests: total number of requests
- Reading: number of reads
- Writing: number of writes
- Waiting: number of keep-alive connections

Here is the source for the extension class.

```
package com.appdynamics.monitors.nginx;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.StringReader;
import java.util.HashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.log4j.Logger;

import com.singularity.ee.agent.systemagent.api.AManagedMonitor;
import com.singularity.ee.agent.systemagent.api.MetricWriter;
import com.singularity.ee.agent.systemagent.api.TaskExecutionContext;
import com.singularity.ee.agent.systemagent.api.TaskOutput;
import com.singularity.ee.agent.systemagent.api.exception.TaskExecutionException;
import com.singularity.ee.util.httpclient.HttpClientWrapper;
import com.singularity.ee.util.httpclient.HttpExecutionRequest;
import com.singularity.ee.util.httpclient.HttpExecutionResponse;
import com.singularity.ee.util.httpclient.HttpOperation;
import com.singularity.ee.util.httpclient.IHttpClientWrapper;
import com.singularity.ee.util.log4j.Log4JLogger;

/**
 * NGinXStatusMonitor is a class that provides metrics on NGinX server by using the
 * NGinX status stub.
 */
public class NGinXMonitor extends AManagedMonitor
{
 /**
  ** The metric prefix indicates the metric's position in the AppDynamics metric hierarchy.
  ** It is prepended to the metric name when the metric is uploaded to the Controller.
  ** These metrics can be found in the AppDynamics metric hierarchy under
  ** Application Infrastructure Performance|{@literal <}Node{@literal >}|Custom
Metrics|WebServer|NGinX|Status
  */
 private final static String metricPrefix = "Custom Metrics|WebServer|NGinX|Status|";

        /* Needed to connect to the Controller. Some environments may also require
credentials. */
 protected volatile String host;
 protected volatile String port;
 protected volatile String location;

        /* Hash map to store metric values obtained from the source, in this example the NGinX
server. */
 private Map<String,String> resultMap = new HashMap<String,String>();

 protected final Logger logger = Logger.getLogger(this.getClass().getName());
```

```
 /**
  ** Main execution method that uploads the metrics to the AppDynamics Controller.
  ** @see com.singularity.ee.agent.systemagent.api.ITask#execute(java.util.Map,
com.singularity.ee.agent.systemagent.api.TaskExecutionContext)
  */
 public TaskOutput execute(Map<String, String> arg0, TaskExecutionContext arg1)
   throws TaskExecutionException
 {
  try
  {
   host = arg0.get("host");
   port = arg0.get("port");
   location = arg0.get("location");

                        /* Gets the values for the metrics and populates the resultsMap. */
   populate();

                        /* Outputs the metrics: metric name, value and processing qualifiers.
*/
   printMetric("Activity|up", 1,
    MetricWriter.METRIC_AGGREGATION_TYPE_OBSERVATION,
    MetricWriter.METRIC_TIME_ROLLUP_TYPE_SUM,
    MetricWriter.METRIC_CLUSTER_ROLLUP_TYPE_COLLECTIVE);

   printMetric("Activity|Active Connections", resultMap.get("ACTIVE_CONNECTIONS"),
    MetricWriter.METRIC_AGGREGATION_TYPE_OBSERVATION,
    MetricWriter.METRIC_TIME_ROLLUP_TYPE_CURRENT,
    MetricWriter.METRIC_CLUSTER_ROLLUP_TYPE_INDIVIDUAL
   );

   printMetric("Activity|Server|Accepts", resultMap.get("SERVER_ACCEPTS"),
    MetricWriter.METRIC_AGGREGATION_TYPE_OBSERVATION,
    MetricWriter.METRIC_TIME_ROLLUP_TYPE_AVERAGE,
    MetricWriter.METRIC_CLUSTER_ROLLUP_TYPE_COLLECTIVE
   );

   printMetric("Activity|Server|Handled", resultMap.get("SERVER_HANDLED"),
    MetricWriter.METRIC_AGGREGATION_TYPE_OBSERVATION,
    MetricWriter.METRIC_TIME_ROLLUP_TYPE_AVERAGE,
    MetricWriter.METRIC_CLUSTER_ROLLUP_TYPE_COLLECTIVE
   );

   printMetric("Activity|Server|Requests", resultMap.get("SERVER_REQUESTS"),
    MetricWriter.METRIC_AGGREGATION_TYPE_OBSERVATION,
    MetricWriter.METRIC_TIME_ROLLUP_TYPE_AVERAGE,
    MetricWriter.METRIC_CLUSTER_ROLLUP_TYPE_COLLECTIVE
   );

   printMetric("Activity|Reading", resultMap.get("READING"),
    MetricWriter.METRIC_AGGREGATION_TYPE_OBSERVATION,
    MetricWriter.METRIC_TIME_ROLLUP_TYPE_AVERAGE,
    MetricWriter.METRIC_CLUSTER_ROLLUP_TYPE_COLLECTIVE
   );

   printMetric("Activity|Writing", resultMap.get("WRITING"),
    MetricWriter.METRIC_AGGREGATION_TYPE_OBSERVATION,
    MetricWriter.METRIC_TIME_ROLLUP_TYPE_AVERAGE,
    MetricWriter.METRIC_CLUSTER_ROLLUP_TYPE_COLLECTIVE
   );

   printMetric("Activity|Waiting", resultMap.get("WAITING"),
    MetricWriter.METRIC_AGGREGATION_TYPE_OBSERVATION,
    MetricWriter.METRIC_TIME_ROLLUP_TYPE_AVERAGE,
    MetricWriter.METRIC_CLUSTER_ROLLUP_TYPE_COLLECTIVE
   );
   /* Upload the metrics to the Controller. */
```

```
   return new TaskOutput("NGinX Metric Upload Complete");
  }
  catch (Exception e)
  {
   return new TaskOutput("Error: " + e);
  }
 }

 /**
  * Fetches Statistics from NGinX Server
  * @throws InstantiationException
  * @throws IllegalAccessException
  * @throws ClassNotFoundException
  * @throws IOException
  */
 protected void populate() throws InstantiationException,
   IllegalAccessException, ClassNotFoundException, IOException
 {
  IHttpClientWrapper httpClient = HttpClientWrapper.getInstance();

  HttpExecutionRequest request = new HttpExecutionRequest(getConnectionURL(), "",
HttpOperation.GET);
  HttpExecutionResponse response = httpClient.executeHttpOperation(request, new
Log4JLogger(logger));

  Pattern numPattern = Pattern.compile("\\d+");
  Matcher numMatcher;

  BufferedReader reader = new BufferedReader(new StringReader(response.getResponseBody()));
  String line, whiteSpaceRegex = "\\s";

  while ((line=reader.readLine()) != null)
  {
   if (line.matches("Active connections"))
   {
    numMatcher = numPattern.matcher(line);
    numMatcher.find();
    resultMap.put("ACTIVE_CONNECTIONS", numMatcher.group());
   }
   else if (line.matches("server"))
   {
    line = reader.readLine();

    String[] results = line.trim().split(whiteSpaceRegex);

    resultMap.put("SERVER_ACCEPTS", results[0]);
    resultMap.put("SERVER_HANDLED", results[1]);
    resultMap.put("SERVER_REQUESTS", results[2]);
   }
   else if (line.contains("Reading"))
   {
    String[] results = line.trim().split(whiteSpaceRegex);
    resultMap.put("READING", results[1]);
    resultMap.put("WRITING", results[3]);
    resultMap.put("WAITING", results[5]);
   }
  }
 }

 /**
  * Returns the metric to the AppDynamics Controller.
  * @param  metricName  Name of the Metric
  * @param  metricValue  Value of the Metric
  * @param  aggregation  Average OR Observation OR Sum
  * @param  timeRollup  Average OR Current OR Sum
  * @param  cluster   Collective OR Individual
```

```
  *
  * Overrides the Metric Writer class printMetric() by building the string that provides all
the fields needed to
  * process the metric.
  */
 public void printMetric(String metricName, Object metricValue, String aggregation, String
timeRollup, String cluster)
 {
  MetricWriter metricWriter = getMetricWriter(getMetricPrefix() + metricName,
   aggregation,
   timeRollup,
   cluster
  );

  metricWriter.printMetric(String.valueOf(metricValue));
 }

 protected String getConnectionURL()
 {
  return "http://" + host + ":" + port + "/" + location;
 }

 protected String getMetricPrefix()
 {
  return metricPrefix;
```

```
  }
}
```

The monitor.xml File

Create a monitor.xml file with a <monitor> element to configure how the machine agent will execute the extension.

1. Set the <name> to the name of your Java monitoring extension class.

2. Set the <type> to "managed".

3. The <execution-style> can be "continuous" or "periodic".

- Continuous means to collect the metrics averaged over time; for example, average CPU usage per minute. In continuous execution, the Machine Agent invokes the extension once and the program runs continuously, returning data every 60 seconds.
- Periodic means to invoke the monitor at a specified frequency. In periodic execution the Machine Agent invokes the extension, runs it briefly, and returns the data on the schedule set by the <execution-frequency-in-seconds> element.

4. If you chose "periodic" for the execution style, set the frequency of collection in <execution-timeout-in-secs> element. The default frequency is 60 seconds. If you chose "continuous" this setting is ignored.

5. Set the <type> in the <monitor-run-task> child element to "java".

6. Set the <execution-timeout-in-secs> to the number of seconds before the extension times out.

7. Specify any required task arguments in the <task-arguments> element. The default arguments that are specified here are the only arguments that the extension uses. They are not set anywhere else.

8. Set the <classpath> to the jar file that contains your extension's classes. Include any dependent jar files, separated by semicolons.

9. Set the <impl-class> to the full path of the class that the Machine Agent invokes.

Sample monitor.xml Files

The following monitor.xml file configures the NGinXMonitor monitoring extension. This extension executes every 60 seconds.

```
<monitor>
        <name>NGinXMonitor</name>
        <type>managed</type>
        <description>NGinX server monitor</description>
        <monitor-configuration></monitor-configuration>
        <monitor-run-task>
                <execution-style>periodic</execution-style>
                <execution-frequency-in-seconds>60</execution-frequency-in-seconds>
                <name>NGinX Monitor Run Task</name>
                <display-name>NGinX Monitor Task</display-name>
                <description>NGinX Monitor Task</description>
                <type>java</type>
                <execution-timeout-in-secs>60</execution-timeout-in-secs>
                <task-arguments>
                        <argument name="host" is-required="true" default-value="localhost" />
                        <argument name="port" is-required="true" default-value="80" />
                        <argument name="location" is-required="true"
default-value="nginx_status" />
                </task-arguments>
                <java-task>
                        <classpath>NginxMonitor.jar</classpath>
                        <impl-class>com.appdynamics.nginx.NGinXMonitor</impl-class>
                </java-task>
        </monitor-run-task>
</monitor>
```

The next monitor.xml file configures the MysqlMonitor. This monitor executes every 60 seconds, has four required task arguments and one optional task argument and one dependent jar file.

```
<monitor>
    <name>MysqlMonitor</name>
    <enabled>true</enabled>
    <type>managed</type>
    <description>Monitors Mysql database system </description>
    <monitor-configuration></monitor-configuration>
    <monitor-run-task>
        <execution-style>periodic</execution-style>
        <execution-frequency-in-seconds>60</execution-frequency-in-seconds>
        <name>Mysql Monitor Run Task</name>
        <display-name>Mysql Monitor Task</display-name>
        <description>Mysql Monitor Task</description>
        <type>java</type>
        <execution-timeout-in-secs>60</execution-timeout-in-secs>
        <task-arguments>
            <argument name="host" is-required="true" default-value="localhost" />
            <argument name="port" is-required="true" default-value="3306" />
            <argument name="user" is-required="true" default-value="root" />
            <argument name="password" is-required="true" default-value="welcome" />

            <!--
            The tier under which the metrics should appear in the metric browser.
            If this argument is left out then the metrics will be registered in every tier.
            -->
           <argument name="tier" is-required="false" default-value="1stTier" />
        </task-arguments>
        <java-task>
            <classpath>mysql.jar;mysql-connector-java-5.1.17-bin.jar</classpath>

<impl-class>com.singularity.ee.agent.systemagent.monitors.database.mysql.MysqlMonitor</impl-class
</java-task>
    </monitor-run-task>
</monitor>
```

Using Your Monitoring Extension as a Task in a Workflow

Your monitoring extension can be invoked as a task in a workflow if you upload the zip file to the task library. Use the instructions in To package the XML files as a Zip archive to upload the Java monitor to the Task Library.