# Writing a HyperGlance Server data collector

HyperGlance server is composed of a number of J2EE components deployed into the JBoss application server.

The HyperGlance server Collector Manager gathers JBoss Microcontainer bean components of type “com.realstatus.hgs.collection.CollectorPluginDescriptorImpl” and then calls back to them during the data collection phase to make their contributions to the overall HyperGlance data model.

A plugin descriptor can be configured to specify:

* One of two modes of topology collection:
  + SQL discovery - queries can be directly placed into the plugin descriptor in order gather nodes, links, endpoints & alarms from an SQL database.
  + Programmatic discovery – A callback class is specified and invoked, allowing discovery of topology elements from third-party APIs.
* Other callback classes may be specified for:
  + Collection of the latest performance data values.
  + Satisfying historical performance data requests for graphing.
  + Data model refinements.

## Topology data collection

The HyperGlance server periodically collects all topological data according to a user-defined cron schedule which, by default, fires every 10 minutes. Two modes of topology collection exist, SQL collectors and programmatic collectors.

### SQL topology collectors

Providing SQL queries to run during the HyperGlance server’s discovery phase involves two beans - one describing the database connection properties and another including the SQL queries.

A bean of type “*com.realstatus.hgs.collection.JdbcDatasource*” describes the database connection properties, as shown below:

<bean name=*"AcmeJdbcDatasource"* class=*"com.realstatus.hgs.collection.JdbcDatasource"*>

<property name=*"jndiName"*>AcmeDS</property>

<property name=*"connectionUrl"*>

jdbc:sqlserver://10.0.1.123.;InstanceName=ACME\_INSTANCE;DatabaseName=ACME\_DB

</property>

<property name=*"driverClass"*>com.microsoft.sqlserver.jdbc.SQLServerDriver</property>

<property name=*"username"*>Admin</property>

<property name=*"password"*>Admin</property>

<property name=*"preparedStatementCacheSize"*>50</property>

</bean>

Figure - database connection properties bean

A “*com.realstatus.hgs.collection.CollectorPluginDescriptorImpl*” bean can be configured to describe an SQL collector by specifying:

* A datasource name used for identifying the collector and any topology elements it contributes
* A list of the SQL queries to execute
* The JDBC connection properties for the database to connect to (the bean for which was described previously).

A skeletal example of a plugin descriptor for an SQL collector is shown below:

<bean name=*"AcmeCollectorPluginDescriptor"* class=*"com.realstatus.hgs.collection.CollectorPluginDescriptorImpl"*>

<property name=*"datasourceName"*>ACME</property>

<property name=*"jdbcDatasource"*>

<inject bean=*"AcmeJdbcDatasource"*/>

</property>

<property name=*"nodeQueries"*>

<list class=*"java.util.ArrayList"* elementClass=*"java.lang.String"*>

<value><!-- SQL query for collecting nodes -->

</value>

</list>

</property>

<property name=*"endpointQueries"*>

<list class=*"java.util.ArrayList"* elementClass=*"java.lang.String"*>

<value><!-- SQL query for collecting endpoints -->

</value>

</list>

</property>

<property name=*"linkQueries"*>

<list class=*"java.util.ArrayList"* elementClass=*"java.lang.String"*>

<value><!-- SQL query for collecting links -->

</value>

</list>

</property>

</bean>

Figure - SQL collector descriptor

A full example, including example queries, is in [Appendix A](#_Appendix_A_–).

Queries can return attributes of any name and type and they will become String properties of the entity in question.

It is mandatory that queries return fields of the following names – either the actual field name or aliased to it (not case sensitive):

**Nodes:** nodeId

(e.g. *SELECT id AS nodeId, … FROM AcmeNode*)

**Endpoints:** endpointId, nodeId

(e.g. *SELECT id AS endpointId, nodeId, … FROM AcmeEndpoint*)

**Links:** linkId, endAId, endBId

(e.g. *SELECT id AS linkId, interface1 AS endAId,*

*interface2 AS endBId FROM AcmeLink*)

HyperGlance server will not allow the construction of a data model that is referentially incoherent – so, for example, if the endpoints query matches a node that is not matched by the nodes query then the endpoint in question will not be created.

### Programmatic topology collectors

A collector can perform programmatic topology data collection by specifying a discovery executor class as the “discoveryClass” property in its plugin descriptor. If a discovery class is defined then the execute() method will be invoked on an instance of that class by the data collection job.

Configuring the “*com.realstatus.hgs.collection.CollectorPluginDescriptorImpl*” bean for a programmatic collector is demonstrated by the following fragment:

<bean name=*"AcmeCollectorPluginDescriptor"* class=*"com.realstatus.hgs.collection.CollectorPluginDescriptorImpl"*>

<property name=*"datasourceName"*>ACME</property>

<property name=*"discoveryClass"*>com.acme.ExampleDiscoveryExecutor</property>

</bean>

Figure - programmatic collector descriptor

The ExampleDiscoveryExecutor referenced in the example snippet must implement the DiscoveryExecutor interface which has the following interface:

/\*\*

\* Interface intended for implementation by Collectors as the callback entry point for programmatic topology data collection.

\*

\* <p> The <code>execute()</code> method on this class is invoked as part of the HyperGlance server's data discovery phase.

\*/

**public** **interface** DiscoveryExecutor {

/\*\*

\* Callback method invoked as part of the HyperGlance server's data discovery phase.

\*

\* **@see** CollectorPluginDescriptor

\* **@see** ModelUpdate

\* **@see** ModelLookup

\*/

**public** **void** execute(CollectorPluginDescriptor pluginDescriptor,

ModelUpdate update, ModelLookup lookup);

}

Figure - DiscoveryExecutor interface

*ModelUpdate* and *ModelLookup* are parameters of the execute method. *ModelUpdate* allows collector writers to contribute model elements to the HyperGlance server. *ModelLookup* allows collector writers to find model elements that are known to the HyperGlance server.

An example programmatic collector that uses the ModelUpdate object to create a two-node system:

public class ExampleDiscoveryExecutor **implements** DiscoveryExecutor **{**

public void execute**(**CollectorPluginDescriptor pluginDescriptor**,**

ModelUpdate update**,** ModelLookup lookup**)** **{**

// add a server node as node1

Map**<**String**,** String**>** node1Attributes **=** **new** HashMap**<**String**,** String**>();**

node1Attributes**.**put**(**"name"**,** "My Server"**);**

update**.**addNode**(**"node1"**,** node1Attributes**);**

// add a router node as node2

Map**<**String**,** String**>** node2Attributes **=** **new** HashMap**<**String**,** String**>();**

node2Attributes**.**put**(**"name"**,** "My Router"**);**

update**.**addNode**(**"node2"**,** node2Attributes**);**

// add an endpoint to node1

Map**<**String**,** String**>** endpoint1Attributes **=** **new** HashMap**<**String**,** String**>();**

endpoint1Attributes**.**put**(**"device"**,** "A Network Interface Card"**);**

update**.**addEndpoint**(**"node1"**,** "endpoint1"**,** endpoint1Attributes**);**

// add an endpoint to node2

Map**<**String**,** String**>** endpoint2Attributes **=** **new** HashMap**<**String**,** String**>();**

endpoint2Attributes**.**put**(**"device"**,** "An Ethernet NIC"**);**

update**.**addEndpoint**(**"node2"**,** "endpoint2"**,** endpoint2Attributes**);**

// add a link between the endpoints

Map**<**String**,** String**>** linkAttributes **=** **new** HashMap**<**String**,** String**>();**

linkAttributes**.**put**(**

"description"**,**

"A connection between our server node and router node"**);**

update**.**addLink**(**"endpoint1"**,** "endpoint2"**,** "link1"**,** linkAttributes**);**

**}**

**}**

Figure - DiscoveryExecutor example

By adding the elements to the ModelUpdate those components will be persisted to the HyperGlance server’s database once the executor has completed.

The identifiers (“foreignSourceId” as it is referred to in our ModelUpdate API) used for each element type (nodes, links, endpoints & alarms) must be unique within each category of element types contributed by a collector. In other words, a collector writer can consider themselves to have their own identification namespace which is indicated by their datasourceName. The foreignSourceIds of the model elements are used to correlate them with elements that are already in the HyperGlance server’s database from previous runs of the executor in order to update those with any new or changed attributes. Any elements that were added by a previous run of the executor but not as part of the most-recent run will be removed from the database automatically. Under these rules the above sample executor will create a stable 2-node, 2-endpoints, 1-link network; it will not grow the network by 2 nodes on each execution because the foreignSourceIds remain the same.

## Performance data collection

The HyperGlance server allows specification of callback classes to collect the latest performance data and to request historical performance data.

### Contributing the latest performance data metric values

The HyperGlance server periodically collects the latest performance data according to a user-defined cron schedule which, by default, fires every 5 minutes. A collector can specify a PerformanceExecutor implementation as its “collectionClass” in a PerformanceDataCollectionProperties bean in its plugin descriptor to hook into events for performance collection.

Example plugin descriptor fragment:

<bean name=*"AcmePerformanceDataCollectionProperties"* class=*"com.realstatus.hgs.collection.PerformanceDataCollectionProperties"*>

<property name=*"collectionClass"*>

com.acme.ExamplePerformanceDataCollector

</property>

<property name=*"collectionEntityBatchSize"*>100</property>

<property name=*"fetchClass"*>com.acme.ExamplePerformanceFetcher</property>

</bean>

Figure - performance collection properties

The ExamplePerformanceDataExecutor must implement PerformanceExecutor which has the following interface:

/\*\*

\* Interface intended for implementation by Collectors as the callback entry point for programmatic performance data collection.

\*

\* <p> The methods on this class are invoked as part of the HyperGlance server's performance data collection phase.

\*/

**public** **interface** PerformanceExecutor {

/\*\*

\* Callback method invoked by the HyperGlance server at the start of the performance data collection phase

\* to produce a number of <code>BatchGroup</code>s which can be divided and processed concurrently by the <code>collect</code> method.

\*

\* **@see** PerformanceExecutor#collect(Long, BatchGroup)

\* **@see** CollectorPluginDescriptor

\* **@see** BatchGroup

\*/

**public** Collection<BatchGroup> getBatchGroups(CollectorPluginDescriptor pluginDescriptor, ModelLookup lookup);

/\*\*

\* Callback method invoked by the HyperGlance server multiple times concurrently during the middle of the performance data collection phase

\* to collect metrics based on the contents of the provided <code>BatchGroup</code>.

\*

\* **@see** BatchGroup

\*/

**public** **void** collect(Long executionTime, BatchGroup batch, ModelLookup lookup);

/\*\*

\* Callback method invoked by the HyperGlance server at the end of the performance data collection phase

\* to attach the collected performance metrics onto the model.

\*

\* **@see** PerformanceModelUpdate

\*/

**public** **void** attach(PerformanceModelUpdate update);

}

Figure - PerformanceExecutor interface

First, getBatchGroups() is invoked in order to give the collector an opportunity to return a set of BatchGroups, each batch-group optionally contains a group of entities that the collector intends to collect metrics from. The entities within each batch-group are further subdivided into smaller batch-groups of a size no greater than that specified by the “collectionEntityBatchSize” property from the PerformanceDataCollectionProperties bean.

Next, the resultant subdivided batch-groups are issued back to the collector as the HyperGlance server makes concurrent calls to the collect() method. The collector is expected to process the provided BatchGroup, accumulating the collected metrics in a thread-safe model of its own.

Finally, after all batch-groups have been processed, the attach() method is invoked and using the PerformanceModelUpdate object a collector can insert their collected metric values to update those held within the HyperGlance server’s internal model.

### Responding to requests for historical performance data

The HyperGlance server responds to requests from HyperGlance clients for historical performance data (to render in charts). A Collector can respond to those requests by specifying a “fetchClass” in a PerformanceDataCollectionProperties bean in its plugin descriptor (see Figure 6 - performance collection properties).

/\*\*

\* Interface intended for implementation by Collectors as the callback entry point for requesting historical performance data.

\*/

**public** **interface** PerformanceFetcher {

/\*\*

\* Invoked immediately before any requests for performance data.

\*

\* <p> This allows the Collector writer to perform any expensive setup steps that there may be in preparation for getting

\* performance data from the source.

\*/

**public** **void** onStart(MetricedEntity entity, ModelLookup lookup);

/\*\*

\* Gets performance related meta-data for the given entity.

\*

\* <p>Returning null (the default implementation) indicates the collector wishes fallback to the <code>PerformanceMetaData</code>

\* that was added to the <code>PerformanceDataCollectionProperties</code> bean.

\* **@param** entity that we want performance data values for

\* **@return** An instance of <code>PerformanceMetaData</code> relating to the entity, or null (see description).

\*/

**public** String[] getDefaults(MetricedEntity entity);

/\*\*

\* Invoked on each request for performance data.

\*

\* **@param** entity that we want performance data values for

\* **@param** metricName of the metric

\* **@param** timePeriod that we want metric values for

\* **@return** a <code>TimedValue[]</code> of metric values

\*/

**public** TimedValue[] getTimeSeries(MetricedEntity entity, String metricName, PeriodEnum timePeriod);

/\*\*

\* Invoked immediately after all requests for performance data in this phase of the collection lifecycle have been performed.

\*

\* <p> This allows the Collector writer to dispose of anything they may have been caching or close any sessions they may have been

\* holding open.

\*/

**public** **void** onFinished(MetricedEntity entity);

}

Figure - PerformanceFetcher interface

## Data refinement steps

Once the HyperGlance Server has completed data collection, a collector plugin can specify data refinement steps to post-process the model before it is made available to a HyperGlance client.

Refinements are executed in three passes and an individual refinement can be placed into one of those three passes. All refinements across all collectors in a pass are guaranteed to have executed before any refinements in a subsequent pass; allowing refinements to have dependencies. It is therefore recommended to use the following convention when deciding which pass to place a refinement in:

* **First-pass:** Place ‘mutative’ refinements in this pass, those that change attributes on existing entities. This allows subsequent passes to benefit from the additional information.
* **Second-pass:** Place ‘additive’ refinements in this pass, those that contribute additional elements to the model such as merge-hints or postulated links.
* **Third-pass:** Place ‘filtering’ refinements in this pass, those that remove model elements or attributes. By placing them in the final pass it prevents removing information that a later refinement might want to use; filters are also most often used to alter the boundaries between different collectors’ topologies which is typically constructed by merge-hints in the second pass.

Example plugin descriptor fragment:

<property name=*"firstPassRefinementClasses"*>

<list class=*"java.util.ArrayList"* elementClass=*"java.lang.String"*>

<value>com.acme.ExampleNodeRefinement</value>

</list>

</property>

Figure - refinement class descriptor

A refinement class must implement RefinementExecutor which has the interface given below.

/\*\*

\* Interface intended for implementation by Collectors as the callback entry point for topology data refinement processing.

\*

\* <p> The <code>execute()</code> method on this class is invoked as part of the HyperGlance server's data refinement phase.

\*/

**public** **interface** RefinementExecutor {

/\*\*

\* Callback method invoked as part of the HyperGlance server's data refinement phase.

\*

\* **@see** CollectorPluginDescriptor

\* **@see** ModelRefinement

\* **@see** ModelLookup

\*/

**public** **void** execute(CollectorPluginDescriptor pluginDescriptor, ModelRefinement modelRefinement, ModelLookup dataModel);

}

Figure - RefinementExecutor interface

A refinement uses a provided ModelRefinement object to update and filter model elements as well as to contribute merge-hints and postulated links. For updating attributes on entity a refinement first modifies the entity’s attribute map and then passes the entity into an appropriate update method of the ModelRefinement.

A complete example of a refinement is given below, it assumes that the collector has previously discovered alarms and given them a traffic-light status attribute. The refinement transforms the traffic light status into the HyperGlance server’s own enumeration of possible alarm severities:

public class AlarmSeverityDeterminer **implements** RefinementExecutor **{**

public void execute**(**

CollectorPluginDescriptor pluginDescriptor**,**

ModelRefinement modelRefinement**,**

ModelLookup dataModel**)** **{**

// get all alarms that were added previously by this collector

Collection**<**Alarm**>** alarms **=** dataModel**.**getAlarms**(**pluginDescriptor**.**getDatasourceName**());**

**for** **(**Alarm alarm **:** alarms**)** **{**

// grab this alarm's attribute map

Map**<**String**,** String**>** attributes **=** alarm**.**getAttributes**();**

// get and remove the alarm's traffic light status

// (red, amber, green status) from the attributes

String trafficLightStatus **=** attributes**.**remove**(**"traffic-light status"**);**

// work out the alarm's severity

AlarmSeverityEnum severity **=** AlarmSeverityEnum**.**UNKNOWN**;**

**if** **(**severity **!=** **null)** **{**

**if** **(**trafficLightStatus**.**equals**(**"red"**))** **{**

severity **=** AlarmSeverityEnum**.**MAJOR**;**

**}**

**else** **if** **(**trafficLightStatus**.**equals**(**"amber"**))** **{**

severity **=** AlarmSeverityEnum**.**WARNING**;**

**}**

**else** **if** **(**trafficLightStatus**.**equals**(**"green"**))** **{**

severity **=** AlarmSeverityEnum**.**INFORMATIONAL**;**

**}**

**}**

// update the severity on the alarm itself

alarm**.**setSeverity**(**severity**);**

// apply the update to the ModelRefinement

modelRefinement**.**updateAlarm**(**alarm**);**

**}**

**}**

**}**

Figure - RefinementExecutor example

## Including properties and specifying collector dependencies

Collectors may include a properties file (standard key=value properties) if they need to. This is achieved by setting the propertiesFileLocation property on the CollectorPluginDescriptor bean; the text of the bean property represents the location, relative to the collector .ear directory, of the properties file. These properties will be loaded on starting the collector and can be accessed through our PropertiesLoader (com.realstatus.hgs.util).

<property name=*"propertiesFileLocation"*>settings.properties</property>

Figure 12 – properties file location

Collectors may also define dependencies on other collectors, or deployment components by use of the “demand” tag. This is a standard JBoss MC mechanism used to enforce deployment order.

<demand>AnotherCollector</demand>

<demand>jboss.jca:service=DataSourceBinding,name=MyDataSource</demand>

Figure 13 – properties file location

# Packaging and deploying a data collector

Simple data collectors which do not need to contribute any Java classes (i.e. those that just specify SQL queries for topology discovery) can be deployed simply by copying their \*–jboss-beans.xml plugin descriptor into *~jboss/server/default/deploy/deploy.last*.

The deployment will be automatically detected and validated for correctness. If validation is successful then it will be added to the HyperGlance Collection Manager’s list of active Collectors.

Data collectors that contribute classes such as Discovery or Refinement classes need to package their plugin descriptor and classes into an ear file (in archive or exploded directory form) and copy that into *~jboss/server/default/deploy/deploy.last/.*

A trivial example of an EAR directory structure is as follows:

exampleCollector.ear/

/META-INF

example-jboss-beans.xml

/com/acme/example/

ExampleRefinement.class

ExampleTopologyDiscovery.class

If you are writing a collector based on the TemplateCollector then the README.txt file in the TemplateCollector describes its contents and how to build and deploy it.

# Full list of plugin descriptor properties

## CollectorPluginDescriptor Bean

|  |  |  |
| --- | --- | --- |
| Property Name | Description | Mandatory? |
| datasourceName | The name of the datasource. This is associated with all collected entities and can be viewed and used in the HyperGlance client (in render profiles etc). | Yes |
| jdbcDatasource | The JDBC datasource bean injected into this bean. See definition below. | No |
| performanceDataCollectionProperties | The PerformanceDataCollectionProperties bean injected into this bean. See definition below. | No |
| performanceMetaData | The PerformanceMetaData bean injected into this bean. See definition below. | No |
| nodeQueries | A list of SQL queries. Each row of the resultset returned represents a Node. | No |
| endpointQueries | A list of SQL queries. Each row of the resultset returned represents an Endpoint. | No |
| linkQueries | A list of SQL queries. Each row of the resultset returned represents a Link. | No |
| alarmQueries | A list of SQL queries. Each row of the resultset returned represents an Alarm. | No |
| discoveryClass | The name of the class to be called during the HyperGlance server data discovery phase. Must implement DiscoveryExecutor. | No |
| firstPassRefinementClasses | A list of classnames. These classes must implement RefinementExecutor. | No |
| secondPassRefinementClasses | A list of classnames. These classes must implement RefinementExecutor. | No |
| thirdPassRefinementClasses | A list of classnames. These classes must implement RefinementExecutor. | No |
| propertiesFileLocation | The location (relative to the collector .ear directory) of the collector’s properties file. | No |

## JdbcDatasource Bean

|  |  |  |
| --- | --- | --- |
| Property Name | Description | Mandatory? |
| jndiName | The JNDI name that the JDBC datasource described by this bean will be bound to so that it can be looked up by the HyperGlance server. This name must be unique amongst all HyperGlance collector plugins – so it would make sense to name it after the collector. | Yes |
| connectionUrl | The URL to connect to the database. | Yes |
| driverClass | The database driver class name. | Yes |
| username | A username with the permissions required to run the specified SELECT statements. | Yes |
| password | The password for the user. | Yes |
| preparedStatementCacheSize | Defaults to 50. | No |

## PerformanceDataCollectionProperties Bean

|  |  |  |
| --- | --- | --- |
| Property Name | Description | Mandatory? |
| collectionClass | The name of the class to be called during the HyperGlance server performance collection phase. Must implement PerformanceExecutor. | No |
| collectionEntityBatchSize | Defaults to 100. | No |
| fetchClass | The name of the class to be called on receiving requests for historical performance data. Must implement PerformanceFetcher. | No |

## PerformanceMetaData Bean

|  |  |  |
| --- | --- | --- |
| Property Name | Description | Mandatory? |
| defaultRollup | The name of the default rollup for performance metric value requests. | No |
| intervalsToPeriods | A mapping between known performance data intervals (time between data samples) and graphing periods so that HyperGlance can select the right interval given a graph period. | No |

# Appendix A – simple plugin descriptor example

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<deployment xmlns=*"urn:jboss:bean-deployer:2.0"*>

<bean name=*"SolarwindsJdbcDatasource"* class=*"com.realstatus.hgs.collection.JdbcDatasource"*>

<property name=*"jndiName"*>SolarwindsDS</property>

<property name=*"connectionUrl"*>

jdbc:sqlserver://<host>;InstanceName=<DBinstance>;DatabaseName=<DBname>

</property>

<property name=*"driverClass"*>

com.microsoft.sqlserver.jdbc.SQLServerDriver

</property>

<property name=*"username"*>Admin</property>

<property name=*"password"*>Admin</property>

<property name=*"preparedStatementCacheSize"*>50</property>

</bean>

<bean name=*"SolarwindsCollectorPluginDescriptor"*

class=*"com.realstatus.hgs.collection.CollectorPluginDescriptorImpl"*>

<property name=*"datasourceName"*>Solarwinds</property>

<property name=*"jdbcDatasource"*>

<inject bean=*"SolarwindsJdbcDatasource"*/>

</property>

<property name=*"nodeQueries"*>

<list class=*"java.util.ArrayList"* elementClass=*"java.lang.String"*>

<value>

SELECT NodeID as nodeid,

Name as "name",

SNMPOID as "OID",

SystemDescription as "System Description",

SNMPDescription as "SNMP Description",

SNMPLocation as "SNMP Location",

OperatingSystem as "Operating System",

PrimaryAddress as "Primary Address",

MachineType as "Machine Type"

FROM DiscoveryNodes

</value>

</list>

</property>

<property name=*"endpointQueries"*>

<list class=*"java.util.ArrayList"* elementClass=*"java.lang.String"*>

<value>

SELECT EndPointID as endpointid,

NodeID as nodeid,

Port as "Port",

Label as "Label"

FROM DiscoveryEndPoints

</value>

</list>

</property>

<property name=*"linkQueries"*>

<list class=*"java.util.ArrayList"* elementClass=*"java.lang.String"*>

<value>

SELECT

link.EdgeID as linkid,

link.Source as endaid,

link.Destination as endbid,

endpoint1.NodeID as nodeaid,

endpoint2.NodeID as nodebid,

link.Label as "Label",

link.Speed as "Speed"

FROM DiscoveryEdges link

JOIN DiscoveryEndPoints endpoint1 ON endpoint1.EndPointID = link.Source

JOIN DiscoveryEndPoints endpoint2 ON endpoint2.EndPointID = link.Destination

</value>

</list>

</property>

<property>settings.properties</property>

<demand>AnotherCollectorOnWhichThisDepends</demand>

<demand>YetAnotherCollectorOnWhichThisDepends</demand>

</bean>

</deployment>

Figure 4 - Minimal Solarwinds SQL collector descriptor