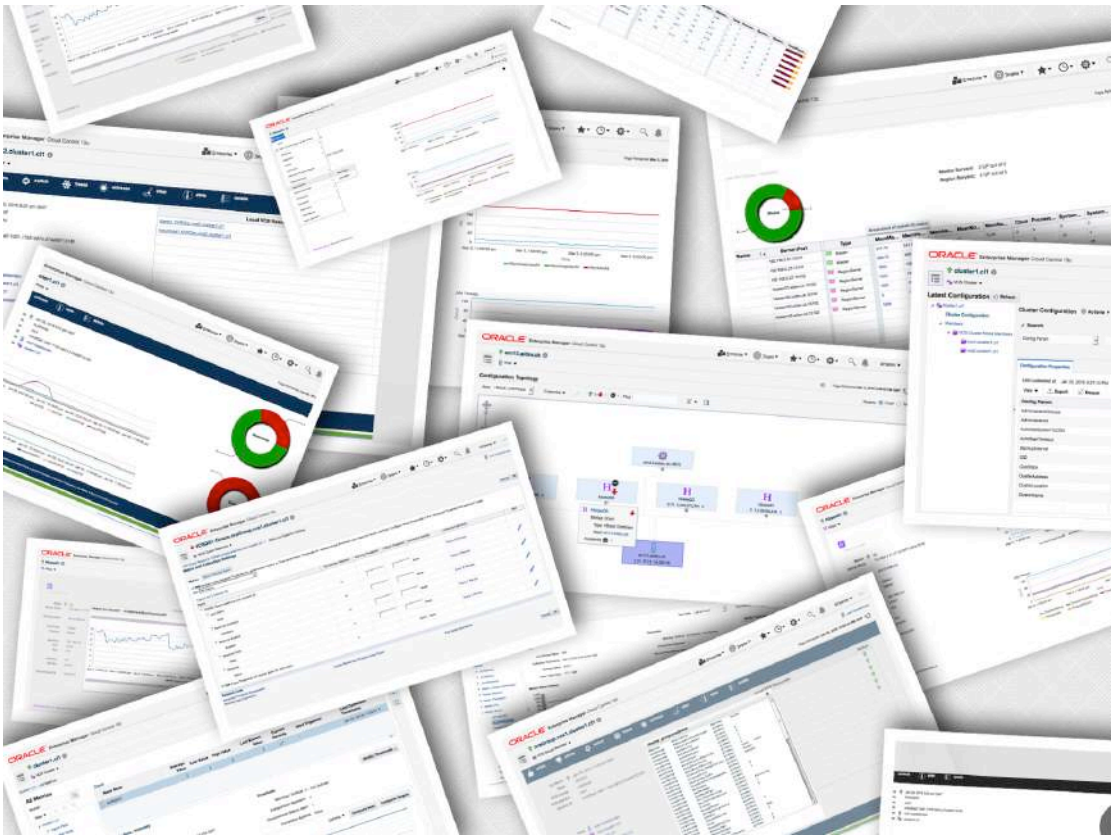
ORACLE ENTERPRISE MANAGER 13C PLUGIN DEVELOPMENT

PRACTICAL TIPS FROM AN EXPERIENCED PLUGIN DEVELOPER

AN AIDEV WHITE PAPER | MAY 2020



wardrop CONSULTING ORACLE | Partner aidev

Revision History

The following changes have been made to this document:

Date	Revision
01.04.2020	Initial draft
18.05.2020	First released version

Credits

A special thanks to Dan Koloski and Sumesh Balakrishnan at Oracle for their review of this paper.

About the Author

AIDEV is a UK-based independent Oracle Enterprise Manager developer operated by Wardrop Consulting Limited, an Oracle Partner company.

It currently has several plugins on the Oracle Enterprise Manager Extensibility Exchange:

- mongoDB
- NGINX
- SSL Certificate
- HBase
- Symantec VCS cluster
- REDIS Data Store

AIDEV has also engaged in the creation of custom plugins for Oracle Enterprise manager customers, developed to meet their specific requirements.

Custom plugins enable seamless integration of non-Oracle supplied target types into the Oracle Enterprise Manager 13c monitoring, alerting, reporting and configuration management frameworks. The plugins are particularly suited to existing Oracle Enterprise Manager customers who need to monitor/alert on new technology but lack the knowledge or experience to do so easily.

AIDEV has been leveraging the Oracle Enterprise Manager Extensibility Development Kit (EDK) to develop custom plugins since Oracle Enterprise Manager 10g.

By using the EDK's powerful plugin development features, AIDEV can deliver enterprise-grade plugins that provide customers with the functionality they require.

Overview

This document details each of the main stages of plugin development using the Oracle Enterprise Manager 13c Extensibility Development Kit (EDK).

Each stage in the process is explained, based on experience gained by plugin developer AIDEV.

The aim of this document is to explain how plugin developers can leverage the EDK to create powerful enterprise-grade custom plugins which seamlessly integrate into Oracle Enterprise Manager 13c.

Assumptions

This document assumes the reader has basic experience of Oracle Enterprise Manager 13c and the Extensibility Development Kit, including the components of a plugin.

The Main Stages of Plugin Development

Oracle Enterprise Manager 13c plugin development is a multi-stage process.

At AIDEV, we perform each of the following distinct stages when developing a new plugin:

1. Identify plugin requirements
2. Source test environments
3. Understand the technology
4. Profile the required targets
5. Identify and define target metrics
6. Develop the backend code
7. Stage the plugin
8. Validate and compile the plugin
9. Import the plugin
10. Deploy to the OMS
11. Deploy to the agent
12. Add a custom target
13. Custom UI development

Each stage identified above is described in detail within the following sections of this document.

By following this process, plugin developers can easily create enterprise-grade plugins for Oracle Enterprise Manager 13c.

Stage 1 – Identify plugin requirements

The first stage in the development process is to identify exact requirements for the new plugin.

For example, these could be a combination of any of the following:

- To monitor and alert on core component status across an environment
- To manage component configuration standards across a site
- To provide single-screen visibility of all components within a cluster
- To allow remote component control (stop/start) from within Oracle Enterprise Manager 13c
- To provide a reporting capability for application availability

If we are developing a custom plugin for a customer, this stage typically involves the plugin developer having meetings with the customer to identify and document all of the main plugin requirements.

The developer may also need to liaise further with technical experts or support staff to gain an understanding of how the target technology works.

Investigation should be performed into how monitoring can be achieved, taking into consideration any metric retrieval interfaces that are published by the technology.

Once the main requirements for the plugin have been identified, the development process can move on to the next stage – sourcing a suitable test environment.

Stage 2 – Source test environments

Obviously this is a very important stage – plugin developers clearly require a test build, sandpit, VM, physical or similar environment to connect to and develop the plugin against.

They will also need to deploy and test the plugin against each Oracle Enterprise Manager supported by the plugin.

Target Environment

It is essential that all major releases and configurations of the custom target(s) are catered for and can be tested against.

This is particularly important for plugins that intend to support multiple target versions. Our mongoDB plugin supports multiple versions of mongoDB (version 2.3 to the current release 4.x) hence requires test environments for each major mongoDB release.

We typically use lightweight virtual machines for test environments as they help facilitate speed of delivery whilst maintaining flexibility in environment build options.

Oracle Enterprise Manager Environment

Plugin development must not be performed against a Production Oracle Enterprise Manager environment. Plugins will have to be deployed multiple times as part of the testing process and failed deployments may require a backout/restoration of the whole Enterprise Manager environment.

It is also essential that all major versions of Oracle Enterprise Manager supported by the plugin are tested against. The EDK version used to create the plugin needs to support all Enterprise Manager versions being tested against. For example, the developer may choose to develop the plugin using EDK v13.3 and certify it against Enterprise Manager 13.3, 13.4.

We strongly recommend using dedicated Enterprise Manager environments for development, ideally hosted on virtual machines to facilitate easy backout/restoration.

Once the required test environments have been sourced, the plugin developer should begin to gain familiarity with the target technology. This is detailed in the next section '*Understand the technology*'.

Stage 3 – Understand the technology

It is extremely important that the plugin developer has a technical understanding of the target technology prior to developing the plugin.

Expert level knowledge is not essential. A fundamental understanding of the technology will however help ensure the plugin is appropriate for purpose.

The developer should assess whether a local or remote Enterprise Manager agent should be used to monitor the target technology. Remote agents allow for easy monitoring of multiple targets running across varied hosts and operating systems, whereas local agents permit the capturing of metrics and running of OS jobs without the need for remote connectivity.

For example, the following steps were required when developing our mongoDB plugin:

- **Learn the core configuration aspects of the technology.**
  Example: MongoDB uses 'sharding' - data distribution across multiple machines. An understanding of mongoDB sharding directly influenced our decision to maintain a single target type for mongod and mongos type instances. This allowed the gathering of common metrics for all instance types whilst providing additional sharding metrics for mongos instances.

- **Understand the security model.**
  Example: Following extensive testing, we were able to identify the roles/privileges needed to retrieve metric information. This in turn defined the configuration required within mongoDB to allow Oracle Enterprise Manager monitoring.

- **Investigate how targets can be monitored programmatically.**
  Example: Various tests were performed against sandpit environments before we opted to use the mongoDB Java drivers for our development. This approach provides the greatest portability and flexibility for our code, whilst permitting remote metric capture and JavaScript based job execution.

If the developer has limited knowledge of the technology, they should aim to learn the required skills, using online materials and testing against sandpit environments.

They may also choose to engage with specialists to further their understanding of the technology.

Once the developer has a sufficient level of technical understanding, they should begin to assess and document the required target(s), as detailed in the next section.

Stage 4 – Profile the required targets

It is essential that a plugin developer understands the target(s) required for a new plugin. Target properties and relationships need to be identified and defined prior to commencing plugin development.

Target properties

Target properties can serve the following purposes:

- They identify each attribute of the custom target
- They provide information to the agent-side plugin code to enable connectivity to the target and perform metric retrieval
- They can permit association between related targets

A custom target will have multiple properties. For our mongoDB plugin, we identified the following properties for the *mongodb_db* target type:

Property Name	Optional	Read Only
servername	N	N
port	N	N
jarloc	N	N
ssl	N	N
sslstore	N	N
mongo_id	Y	N

These properties map directly to the inputs provided when manually adding a mongoDB target into Enterprise Manager:

Security credentials

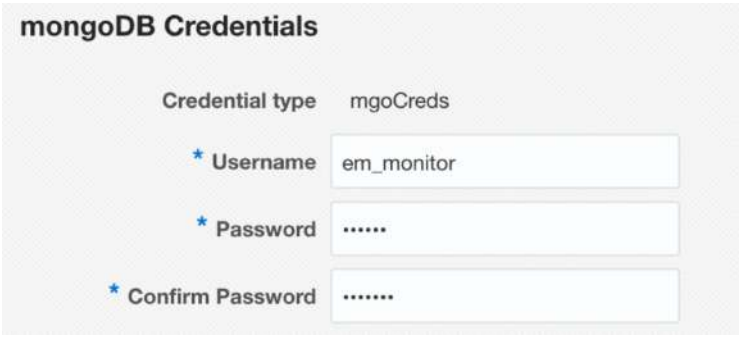
Optionally, security credentials may also be required to allow the backend code to communicate with the managed component.

Monitoring credentials normally map to target properties, initially supplied when the target is added.

Our mongoDB plugin uses the following properties to authenticate against a dedicated monitoring account to retrieve metric data:

Property Name	Optional	Read Only
Username	N	N
Password	N	N

These properties map directly to inputs provided when adding in new mongoDB targets and result in security credentials of type 'mgoCreds' being created in Enterprise Manager:



Plugin developers must identify and document the required target properties/credentials for each target type.

Target relationships

Some plugins may require multiple related target types, whilst other plugins might contain only one target type.

Our VCS plugin contains the following related target types:

Target Type	Relationship
vcs_cluster	Top level cluster target – can contain 1..x vcs_node targets
vcs_node	Node level target – can contain 1..x vcs_node_group targets Member of vcs_cluster
vcs_node_group	VCS group target per node – can contain 1..x vcs_resource targets
vcs_group	VCS group target- cluster type target containing 1..x vcs_node_group targets Provided by vcs_cluster target
vcs_resource	Bottom level target – maps to cluster resource in VCS Member of vcs_node_group target

Target relationships aid the use of core Enterprise Manager features such as Topology Viewer and problem root cause analysis.

If related target types are required, developers should identify and document all required target types and relationships during this stage.

Once the developer has identified and documented the required target properties, security credentials and relationships, this data will input into subsequent stages of the development process.

At this point, the developer should then move on to the next stage in the process – metric definition.

Stage 5 – Identify and define target metrics

A custom target type will require various metrics to determine component status, performance and configuration.

We find that at this stage it helps to identify and document the metrics that will be implemented, their data types and any meaningful thresholds that can serve as a basis for alerting.

Metrics can then be added into the required XML files within the plugin stage directory tree (see Stage 7, '*Stage the plugin'*). Each defined metric will also map to programmatic logic developed within the backend code (see Stage 6, *'Develop the backend code'*).

This stage should be performed in conjunction with the next two stages. New metrics should be identified, corresponding logic developed in the backend code to allow agent retrieval of the required metric data, and finally metric definition and collection behavior added to the staged plugin XML files.

Agent Side Metrics

Traditional Enterprise Manager metrics are collected by the management agent, then uploaded to the OMS. Most metrics within a plugin will be of this type.

The first metric to define, and the only one which is mandatory, is **Response**.

This metric is required for all non-cluster target types and determines one column, **Status**, indicating the current target availability.

For our mongoDB plugin, the Response metric is documented as follows:

Metric Name	Col1	Collection Frequency
Response	Status [NUMBER]	Every 1 min

The next step a plugin developer should take would be to develop backend code to retrieve the Response metric (see Stage 6).

XML content for this metric can then be added to the staged target metadata and collection files(Stage 7).

For our mongoDB plugin, the *metadata* XML entry for the **Response** metric is as follows:

```
<Metric NAME="Response" TYPE="TABLE">
        <Display>
                <Label NLSID="mg_resp">Response</Label>
        </Display>
        <TableDescriptor>
                <ColumnDescriptor NAME="Status" TYPE="NUMBER">
                        <Display>
                                <Label NLSID="mg_status">Status</Label>
                        </Display>
                </ColumnDescriptor>
        </TableDescriptor>
        <QueryDescriptor FETCHLET_ID="OSLineToken">
                <Property NAME="perlBin" SCOPE="SYSTEMGLOBAL">perlBin</Property>
                <Property NAME="scriptsDir" SCOPE="SYSTEMGLOBAL">scriptsDir</Property>
                <Property NAME="ENVMGO_PORT" SCOPE="INSTANCE">port</Property>
                <Property NAME="ENVMGO_HOST" SCOPE="INSTANCE">servername</Property>
                <Property NAME="command" SCOPE="GLOBAL">/bin/bash %scriptsDir%/mongo.sh response</Property>
                <Property NAME="startsWith" SCOPE="GLOBAL">em_result=</Property>
                <Property NAME="delimiter" SCOPE="GLOBAL">=</Property>
<Property NAME="ENVMGO_USERNAME" SCOPE="INSTANCE" OPTIONAL="TRUE">Username</Property>
<Property NAME="ENVMGO_PASSWORD" SCOPE="INSTANCE" OPTIONAL="TRUE">Password</Property>
                <Property NAME="ENVMGO_JARLOC" SCOPE="INSTANCE">jarloc</Property>
<Property NAME="ENVMGO_SSL" SCOPE="INSTANCE" OPTIONAL="TRUE">ssl</Property>
<Property NAME="ENVMGO_SSLSTORE" SCOPE="INSTANCE" OPTIONAL="TRUE">sslstore</Property>
        </QueryDescriptor>
</Metric>
```

Once the Response metric has been identified, the backend code developed and the XML content added, the plugin developer should move on through each additional metric in turn.

A target type will normally contain many additional metrics – these could be single-column or multi-column, single-row or multi-row, depending on the underlying data.

In the case of our mongoDB plugin, one example of a multi-column additional metric is the serverStatus **memory** metric.

This metric captures two columns – **Attribute** (the key value) and **Value**.

Data is collected every 15 minutes by default.

This metric is defined as follows:

Metric Name	Col1	Col2	Collection Frequency
Memory	Attribute [STRING] key	Value [NUMBER]	Every 15 min

The resultant *metadata* XML to be fed into Stage 7 for this metric is:

```
<Metric NAME="memory" TYPE="TABLE">
<Display>
<Label NLSID="NLS_METRIC_mongodb_databasememory">Server Status: Memory</Label>
<Description NLSID="NLS_DESCRIPTION_mongodb_databasememory"> </Description>
</Display>
<TableDescriptor>
<ColumnDescriptor NAME="Attribute" TYPE="STRING" IS_KEY="TRUE">
<Display>
<Label NLSID="NLS_COLUMN_mongodb_databasememoryAttribute">Attribute</Label>
</Display>
</ColumnDescriptor>
<ColumnDescriptor NAME="Value" TYPE="NUMBER">
<Display>
<Label NLSID="NLS_COLUMN_mongodb_databasememoryValue">Value</Label>
</Display>
</ColumnDescriptor>
</TableDescriptor>
<QueryDescriptor FETCHLET_ID="OSLineToken">
<Property NAME="command" SCOPE="GLOBAL">/bin/bash %scriptsDir%/mongo.sh memory</Property>
<Property NAME="scriptLoc" SCOPE="GLOBAL" OPTIONAL="TRUE">%scriptsDir%</Property>
<Property NAME="delimiter" SCOPE="GLOBAL" OPTIONAL="TRUE">|</Property>
                        <Property NAME="ENVMGO_PORT" SCOPE="INSTANCE">port</Property>
                        <Property NAME="ENVMGO_HOST" SCOPE="INSTANCE">servername</Property>
        <Property NAME="ENVMGO_USERNAME" SCOPE="INSTANCE" OPTIONAL="TRUE">Username</Property>
        <Property NAME="ENVMGO_PASSWORD" SCOPE="INSTANCE" OPTIONAL="TRUE">Password</Property>
                        <CredentialRef NAME="monCreds">mongoCredsMonitoring</CredentialRef>
                        <Property NAME="ENVMGO_JARLOC" SCOPE="INSTANCE">jarloc</Property>
        <Property NAME="ENVMGO_SSL" SCOPE="INSTANCE" OPTIONAL="TRUE">ssl</Property>
        <Property NAME="ENVMGO_SSLSTORE" SCOPE="INSTANCE" OPTIONAL="TRUE">sslstore</Property>
</QueryDescriptor>
</Metric>
```

Note how the ‘command’ property runs the backend code, mongo.sh, passing in a parameter ‘memory’ - this returns metric data in the following format:

```
virtual|1645
bits|64
mappedWithJournal|0
mapped|0
resident|30
```

The corresponding *collection* XML for this metric is:

```
<CollectionItem NAME="memory" UPLOAD="YES">
        <Schedule>
                <IntervalSchedule INTERVAL="15" TIME_UNIT="Min"/>
        </Schedule>
        <MetricColl NAME="memory">
                <Condition COLUMN_NAME="Value" OPERATOR="GT" OCCURRENCES="1"
                     MESSAGE="The value of %columnName% for %keyValue% is %value%"
                     MESSAGE_NLSID="EMAGENT_DEFAULT_MESSAGE_WITH_KEY"
                     CLEAR_MESSAGE="Alert for %columnName% for %keyValue% is cleared"
                     CLEAR_MESSAGE_NLSID="EMAGENT_DEFAULT_NO_ROW_CLEAR_MESSAGE_WITH_KEY"/>
        </MetricColl>
</CollectionItem>
```

Consideration should also be given to using advanced metric columns. These are based on existing and previous values collected by the agent and are particularly useful for calculating rate/delta based pseudo values.

We use an advanced metric column within our mongoDB plugin to calculate the delta value for **deletes** based on the difference between current and previous measurements.

This is the XML content within the target *metadata* file:

```
<ColumnDescriptor NAME="Delta_delete" TYPE="NUMBER" COMPUTE_EXPR="Delete - _Delete">
<Display>
<Label NLSID="NLS_COLUMN_mongodb_rdatabase_delt_del">Delete delta</Label>
</Display>
</ColumnDescriptor>
```

Metric alerting thresholds also need to be identified. For our mongoDB plugin, the replication lag metric should provide the ability to alert when the collected lag is greater than a given threshold.

This is the resultant *collection* XML:

```
<CollectionItem NAME="replLag" UPLOAD="YES">
        <Schedule>
                <IntervalSchedule INTERVAL="15" TIME_UNIT="Min"/>
        </Schedule>
        <MetricColl NAME="replLag">
                <Condition COLUMN_NAME="Lag" OPERATOR="GT" OCCURRENCES="1"
                        MESSAGE="The value of %columnName% is %value%"
                        MESSAGE_NLSID="EMAGENT_DEFAULT_MESSAGE_WITH_KEY"
                        CLEAR_MESSAGE="Alert for %columnName% is cleared"
                        CLEAR_MESSAGE_NLSID="EMAGENT_DEFAULT_NO_ROW_CLEAR_MESSAGE_WITH_KEY"
                />
        </MetricColl>
</CollectionItem>
```

Most metrics will only be collected when a target is up, however this is not always the case (for example, log file content metrics).

The following *collection* XML example from our VCS plugin illustrates a metric **GroupState** that is also collected when the target is down:

```
<CollectionItem NAME="GroupState" UPLOAD_ON_FETCH="TRUE" COLLECT_WHEN_DOWN="TRUE">
<Schedule>
<IntervalSchedule INTERVAL="5" TIME_UNIT="Min"/>
</Schedule>
<Condition COLUMN_NAME="group_state" OPERATOR="EQ"/>
</CollectionItem>
```

Repository Side Metrics

In addition to standard agent side metrics, repository-side metrics may also be required.

Repository side metrics capture data from the Oracle Enterprise Manager repository metric data by running custom SQL. They are particularly useful when multi-target aggregate data or summary data is required for a metric.

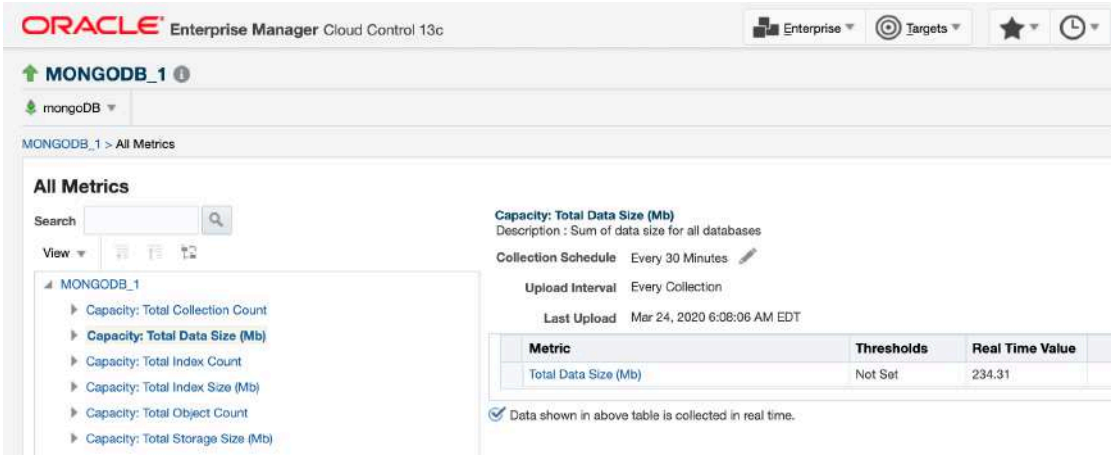
To calculate the total size of all databases within a mongoDB target, our plugin uses a repository-side metric running the following SQL:

```
select target_guid, total as TOT from
(
    select target_guid, sum(value) as total
    from mgmt$metric_current where metric_name = 'dbStats'
    and metric_column = 'dataSizeMb'
    group by target_guid
)
```

This translates into the following *metadata* XML for Stage 7:

```
<Metric NAME="OMR_MGO_DB_DATA_SIZE" TYPE="TABLE" REPOSITORY="TRUE">
<Display>
<Label NLSID="NLS_METRIC_mongodb_dbOMR_MGO_DB_DATA_SIZE">Capacity: Total Data Size (Mb)</Label>
<Description NLSID="NLS_DESCRIPTION_mongodb_dbOMR_MGO_DB_DATA_SIZE">Sum of data size for all databases</Description>
</Display>
<TableDescriptor>
<ColumnDescriptor NAME="TOT" TYPE="NUMBER">
<Display>
<Label NLSID="NLS_COLUMN_mongodb_dbOMR_MGO_DB_DATA_SIZETOT">Total Data Size (Mb)</Label>
</Display>
</ColumnDescriptor>
</TableDescriptor>
<QueryDescriptor FETCHLET_ID="REPOSITORY_SQL">
<Property NAME="Type" SCOPE="GLOBAL">SQL</Property>
<Property NAME="Source" SCOPE="GLOBAL">select target_guid, total as TOT from
(select target_guid, sum(value) as total from mgmt$metric_current where metric_name = 'dbStats'
and metric_column = 'dataSizeMb'
group by target_guid
)</Property>
</QueryDescriptor>
</Metric>
```

Repository side metrics are rendered alongside agent side metric in an identical manner:



Configuration collection

Configuration collection metrics are another consideration - data collected from these metrics can be viewed in Oracle Enterprise Manager under the 'Configuration' menu option.

If a custom target type exposes configuration data, this can be defined as a configuration collection and collected regularly.

Configuration collection metrics need to be marked **CONFIG=TRUE** in the metadata xml and require a custom table to be defined within the plugin to hold the data.

In the case of our mongoDB plugin, we capture build configuration for each target on a daily basis.

Two columns, **ATTRIB** and **SETTING** are captured by the backend code and stored as configuration data in the Enterprise Manager repository:

Metric Name	Col1	Col2	Collection Frequency
MONGODB_BUILDINFO2	Attribute [STRING] key	Value [STRING]	Every 1 day

The target *metadata* xml file contains the following entry for this metric:

```
<Metric NAME="MONGODB_BUILDINFO2" TYPE="RAW" CONFIG="TRUE">
        <Display>
                <Label NLSID="mgo_buildinfo2">mongoDB Build Info</Label>
        </Display>
        <TableDescriptor TABLE_NAME="MONGODB_BUILDINFO2">
                <ColumnDescriptor NAME="ATTRIB" COLUMN_NAME="ATTRIB" TYPE="STRING" IS_KEY="TRUE">
                        <Display>
                                <Label NLSID="attr_label">Attribute</Label>
                        </Display>
                </ColumnDescriptor>
                <ColumnDescriptor NAME="SETTING" COLUMN_NAME="SETTING" TYPE="STRING">
                        <Display>
                                <Label NLSID="sett_label">Value</Label>
                        </Display>
                </ColumnDescriptor>
        </TableDescriptor>
        <QueryDescriptor FETCHLET_ID="OSLineToken">
                <Property NAME="perlBin" SCOPE="SYSTEMGLOBAL">perlBin</Property>
                <Property NAME="scriptsDir" SCOPE="SYSTEMGLOBAL">scriptsDir</Property>
                <Property NAME="ENVMGO_PORT" SCOPE="INSTANCE">port</Property>
                <Property NAME="ENVMGO_HOST" SCOPE="INSTANCE">servername</Property>
<Property NAME="ENVMGO_USERNAME" SCOPE="INSTANCE" OPTIONAL="TRUE">Username</Property>
<Property NAME="ENVMGO_PASSWORD" SCOPE="INSTANCE" OPTIONAL="TRUE">Password</Property>
                <Property NAME="command" SCOPE="GLOBAL">/bin/bash %scriptsDir%/mongo.sh buildinfo</Property>
                <Property NAME="delimiter" SCOPE="GLOBAL">|</Property>
                <Property NAME="ENVMGO_JARLOC" SCOPE="INSTANCE">jarloc</Property>
<Property NAME="ENVMGO_SSL" SCOPE="INSTANCE" OPTIONAL="TRUE">ssl</Property>
<Property NAME="ENVMGO_SSLSTORE" SCOPE="INSTANCE" OPTIONAL="TRUE">sslstore</Property>
                <CredentialRef NAME="monCreds">mongoCredsMonitoring</CredentialRef>
        </QueryDescriptor>
</Metric>
```

A matching entry in the *collection* xml file enforces daily collection of the metric:

```
<CollectionItem NAME="mongdb_buildinfo2_snap" UPLOAD_ON_FETCH="TRUE" CONFIG="TRUE" COLLECT_WHEN_DOWN="FALSE">
<Schedule OFFSET_TYPE="INCREMENTAL">
<IntervalSchedule INTERVAL="24" TIME_UNIT="Hr"/>
</Schedule>
<MetricColl NAME="MONGODB_BUILDINFO2" />
</CollectionItem>
```

This corresponds to an entry in the *snapshotlive* xml file, detailing the custom table and UI labels to use:

```
<METADATA SNAP_TYPE="mongdb_buildinfo2_snap" TARGET_TYPE="mongodb_db" VER="1.0">
     <METADATA_UI_NAME>Database Configuration</METADATA_UI_NAME>
     <TABLE NAME="MONGODB_BUILDINFO2" SINGLE_ROW="N">
          <UI_NAME>Build Info</UI_NAME>
          <COLUMN NAME="ATTRIB" TYPE="STRING" TYPE_FORMAT="1024" IS_KEY="Y">Config Param</COLUMN>
          <COLUMN NAME="SETTING" TYPE="STRING" TYPE_FORMAT="1024" IS_KEY="N">Value</COLUMN>
     </TABLE>
 </METADATA>
```

When the plugin is imported into Oracle Enterprise Manager, we get a custom table created for the configuration data:

```
SQL>desc MONGODB_BUILDINFO2
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
-----------
 ECM_SNAPSHOT_ID                           NOT NULL RAW(16)
 ATTRIB                                    NOT NULL VARCHAR2(1024)
 SETTING                                            VARCHAR2(1024)
```

The table underpins the mongoDB configuration data rendered within Oracle Enterprise Manager:

Config Param	Value
maxBsonObjectSize	16777216
modules	[]
ok	1.0
openssl.compiled	OpenSSL 1.0.1f 6 Jan 2014
openssl.running	OpenSSL 1.0.1f 6 Jan 2014
operationTime.$timestamp.i	1
operationTime.$timestamp.t	1575407765
storageEngines	["devnull","ephemeralForTest","mmapv1","wiredTiger"]

Once a metric has been identified and defined, the plugin developer should move on to the next stage, backend code development.

Stage 6 – Develop the backend code

The next stage in the plugin development process, following metric definition, is to create the method of communication between the Oracle Enterprise Manager agent and the custom target – the *'backend code'*.

The primary focus of this stage is to create a communication and metric retrieval mechanism. This will ensure that the metric data is captured and returned to the agent in an appropriate format.

Oracle Enterprise Manager can monitor targets in a variety of ways, for example remote or local, through perl scripts, bash scripts or Java RMI. If the agent can communicate with the target to capture the required metrics and return the metric data in a way that Enterprise Manager can understand, the list is practically endless.

At AIDEV, we begin by creating standalone scripts, passing in appropriate values for expected target properties and evaluating the retrieved output.

As mentioned previously, the **Response** metric should be targeted first – this governs target status and needs to return a number, either '1' (up) or '0' (down), to the agent.

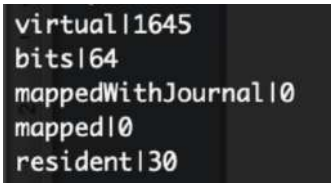
In this case of our mongoDB plugin, the captured metric data is prefixed with *'em_result='* - expected output from the backend code for an 'Up' target is:

```
em_result=1
```

As each additional metric is defined, the backend code should be enhanced to capture the required data.

Multi-column metrics will require a returned payload separated by a delimiter– this is usually a pipe symbol.

As indicated earlier, for our mongoDB plugin the backend script output for the **memory** metric is in the following format:



Appropriate error handling is also required in the backend code to ensure error-free data is passed to the agent.

Within our mongoDB plugin java source code, we handle connection timeouts in the following manner, passing '0' back for the response metric if a timeout is encountered:

```
/* if cannot connect */
catch (com.mongodb.MongoTimeoutException e)
{
 String action = args[0];
 if(action.equals("response"))
 {
   System.out.println("em_result=0");
 }
}
```

Each metric should map to logic within the backend code and be defined with a *command* property in the staged target *metadata* XML (see next section).

For example, in the case of our mongoDB plugin, the *asserts* metric call is:

```
<Property NAME="command"
SCOPE="GLOBAL">/bin/bash %scriptsDir%/mongo.sh
asserts</Property>
```

We have found that test harnesses help to exhaustively test each metric being called by the backend code. They also allow for output comparison when running against varied configurations and versions.

An example of this is the following script, used to test each mongoDB metric within our plugin:

```
#!/bin/bash
# script to test metric operation for a mongoDB target
# one input param - the metric name
#
# Copyright aidev apr 15
#
# V1.1

# 1. place this script in the mongoDB scripts directory within the agent home. cd to this location

# 2. edit the following vars to suit your environment
export MGO_SSL=FALSE
export MGO_SSLSTORE=NONE
export MGO_PORT=21070
export MGO_HOST=192.168.0.53
export MGO_PASSWORD=password
export MGO_USERNAME=em_monitor
export MGO_JARLOC=/home/oracle/mongojars
export JAVA_HOME=/oem/agent/core/12.1.0.3.0/jdk


# 3. run the script as follows:
#
# test the response metric
# sh ./metric_tester.sh response



# code
export CLASSPATH=`pwd`:$CLASSPATH
export dir=`pwd`

$JAVA_HOME/bin/java -Xmx4m -Xms2m -cp "$MGO_JARLOC/mongo-java-driver-2.13.1.jar:$MGO_JARLOC/json-20140107.jar:$dir" mongo_plugin $1
```

Once the backend code has been developed for a specific metric, the developer should move to the next stage, '**Stage the plugin'**, adding metric definition and collection XML into the required plugin files.

Stage 7 – Stage the plugin

The next stage in the process is to gather all information and scripts from the previous stages and bundle them together into a skeleton plugin.

The most straightforward approach to take is to develop metrics individually, performing stages 5, 6,7 and 8 each time.

Normally at this stage, we don't include any custom UI content – this is added later in the process.

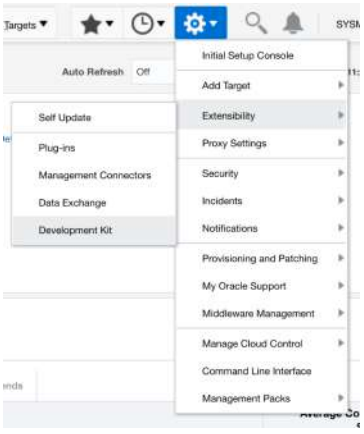
A prerequisite to performing this stage is to install the Enterprise Manager Extensibility Development Kit (EDK) and configure the plugin staging directory structure and XML files.

> It should be noted that recent versions of the EDK reflect the move from Adobe Flex to JET for UI development. For this reason, we recommend using the 13.2 EDK u170321 or later with patch 25453518 – this maintains plugin support for Enterprise Manager 13.2 and above whilst permitting a JET based UI.
>
> All examples in this document are based on EDK 13.2 – we use this version for our mongoDB plugin to ensure compatibility with Enterprise Manager 13.2 and above.

EDK install & staging area creation

The sample plugin code contained within the Enterprise Manager EDK zip file is a good starting point to begin with when creating the staging area.

The EDK can be obtained from the following menu in Oracle Enterprise Manager:



Deployment

- Download the Extensibility Development Kit to your workstation.
- Set your JAVA_HOME environment variable and ensure that it is part of your PATH. You must be running Java 1.7.0_111 or greater. For example:
  - setenv JAVA_HOME /usr/local/packages/j2sdk1.7.0_111
  - setenv PATH $JAVA_HOME/bin:$PATH
- Unzip the downloaded zip file to your local system. For example:
  - Unzip 13.2.0.0.0_edk_partner.zip

  It will create a bin directory under the directory where you have unzipped
- Change to bin directory and run empdk help (for example "empdk -help") from command line, for more details on empdk verbs

Download the EDK from this page and follow the configuration instructions provided.

Create the plugin staging area by copying the location `samples/plugins/HostSample/demo_hostsystem/opar/stage` and amending the content as follows:

`plugin.xml`
- XML to detail plugin metadata
- edit and amend according to plugin definition

`agent/plugin_registry.xml`
- XML to detail metadata and files within the plugin
- edit and amend according to plugin definition

`agent/metadata/`*`target_type`*`.xml`
- the main target XML file detailing target properties and metrics
- add in target metric XML as identified in Stage 5

`agent/default_collection/`*`target_type`*`.xml`
- metric collection behavior
- add in collection XML as identified in Stage 5

`agent/scripts`
- location for each backend script developed in Stage 6
- copy each script into here

`agent/discovery`
- leave this location empty for now as target discovery is beyond the scope of this example

`oms/metadata/assoc`
`oms/metadata/derivedAssocs`
`oms/metadata/discovery`
`oms/metadata/snapshotlive`
`oms/metadata/systemStencil`
`oms/metadata/systemUiIntegration`

- leave these locations empty as the features are beyond the scope of this example

`oms/metadata/targetType/`*`target_type`*`.xml`

- this is typically identical to the agent equivalent

`oms/metadata/default_collection/`*`target_type`*`.xml`

- this is typically identical to the agent equivalent

`oms/metadata/mpcui/`*`target_type`*`.xml`

- this would typically hold the mpcui UI definition, page content and menu layout prior to implementing a JET UI
- as a starting point, create a minimum content xml file based on the supplied sample projects

Note: It is extremely important that the plugin developer defines the ***AgentCompatibility*** tag within the staged plugin.xml file.

This ensures that backward compatibility can be achieved between the newly deployed OMS-side plugin and older agent-side plugin versions.

This is important when upgrading to a newer plugin release.

For our mongoDB plugin version 13.2.0.1.0 file, we define the following:

```
<AgentSideCompatibility>
   <Version>12.1.0.9.0</Version>
   <Version>12.1.0.10.0</Version>
</AgentSideCompatibility>
```

The above entry ensures that agent-side plugin versions 12.1.0.9.0 and 12.1.0.10.0 can work against the OMS-side 13.2.0.1.0 plugin.

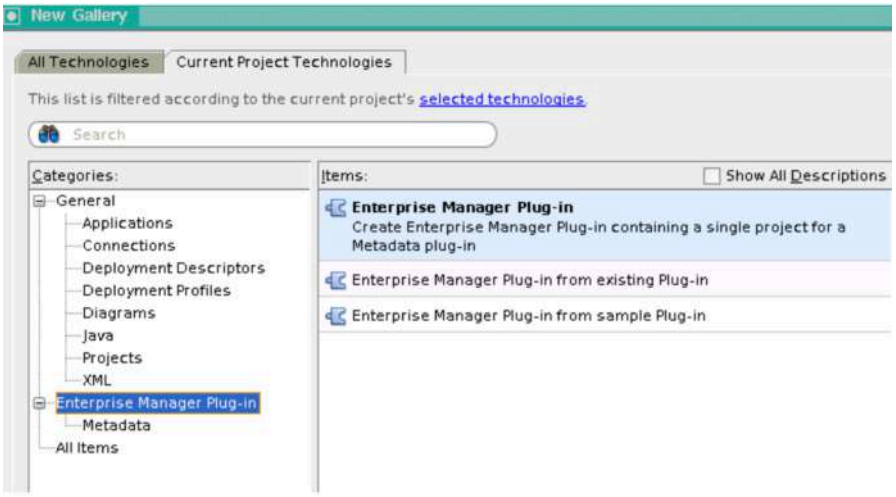
Please refer here for further information on the required XML file content in each of the required files.

XML content creation

The Oracle Enterprise Manager EDK contains multiple sample projects. At AIDEV we recommend using these as a starting point for XML file creation and a point of reference.

Oracle also provides a java-based tool, Plug-in Builder, for generating plugin XML files. This can be particularly helpful for plugin developers with limited experience of plugin development.

We normally develop our plugins manually, using previous plugin content as a starting point. We do however recommend Plug-in Builder for first time plugin developers to gain familiarity with the process.



Plug-in Builder is outside the scope of this document, however more information can be found at https://docs.oracle.com/cd/cloud-control-13.3/EMPRF/GUID-6A94EE77-D7AA-4A30-83AA-B627C41D7264.htm#EMPRF12922.

As a developer creates new plugins, existing code can be reused from previous builds/plugins.

We recommend adding each metric individually, each time performing a plugin validation as detailed in the next stage to troubleshoot any issues.

Stage 8 – Validate and compile the plugin

The **empdk** utility is bundled within the Oracle Enterprise Manager EDK.

It is used to compile an opar file from the plugin source directory structure created in the previous stage.

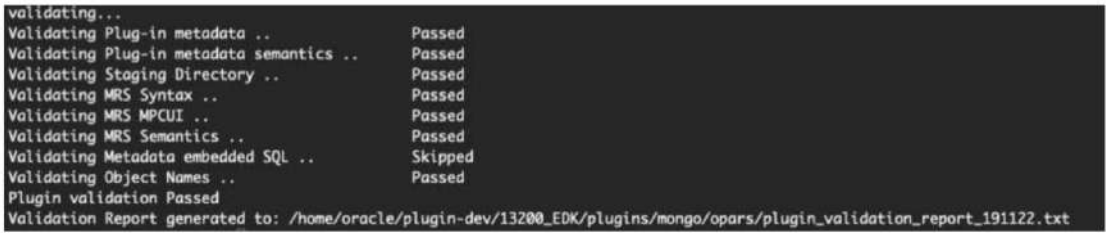
The opar file can then be imported into Enterprise Manager and deployed to the OMS and agent tiers.

Empdk also allows validation of the plugin content, checking for errors and highlighting areas requiring further attention.

To run empdk validation, the command is:

```
empdk validate_plugin -stage_dir {stage location} -out_dir
{opar location} -debug {debug log location}/debug.log
```

Sample output from a successful validation would be:



Stages 7 and 8 should be performed in small increments, adding new metrics, fixing issues and re-validating the plugin.

Once all metrics have been added and final validation is complete for a plugin, it can be compiled into an opar file:

```
empdk create_plugin -stage_dir {stage location} -out_dir {opar
location} -debug {debug log location}/debug.log
```

Sample output:

Important: The plugin version defined in the staged plugin.xml and agent/plugin_registry.xml files must match the version of EDK being used, and in turn the version of Enterprise Manager.

Any attempt to run empdk against a different version of plugin will result in the following error:

Once the opar file has been created, it can be imported into Oracle Enterprise Manager and deployed for further testing.

Stage 9 – Import the plugin

In this stage, the developer will import the plugin opar file created in the previous section.

At AIDEV, we recommend using a non-Production Enterprise Manager system for plugin development – failures in plugin deployment can require full OMS or OMR restoration to resolve.

Virtual machine hosted environments allow for a quick reversal of plugin deployment hence should be considered for the Oracle Enterprise Manager system.

The plugin opar can be imported into Enterprise Manager in the standard way:

```
emcli import_update -file={full path to opar file} -omslocal
```

This imports the plug-in into the Enterprise Manager environment and makes it visible within the console.

Example:

```
$ emcli import_update -
file="/home/oracle/13.2.0.1.0_aidev.mongo.xdbs_2000_0.opar" -omslocal


Processing update: Plug-in - Aidev mongoDB system monitoring plugin
for Oracle Enterprise Manager
Successfully uploaded the update to Enterprise Manager. Use the Self
Update Console to manage this update.
```

Once the plugin is imported into Enterprise Manager, it should be deployed to the OMS and agent tiers.

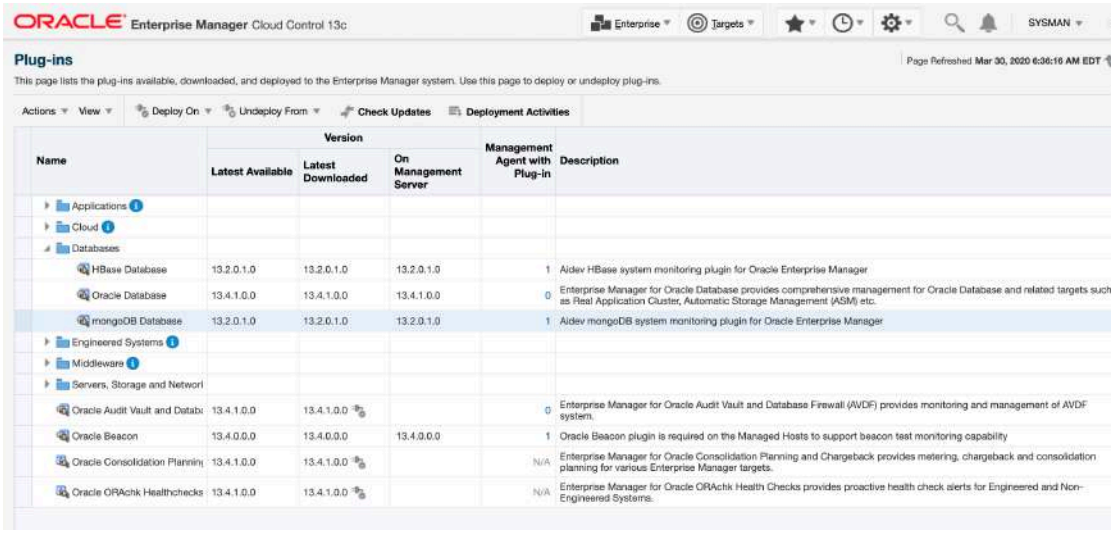
Stage 10 – Deploy to the OMS

Deploy the plugin to the OMS tier using the standard procedure.

Stage 11 – Deploy to the agent

Deploy the plugin to the management agent using the standard procedure.

At this point, the plugin should be visible within Oracle Enterprise Manager and showing as being deployed to a management agent:



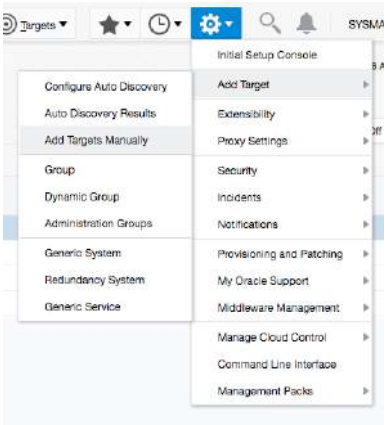
The plugin developer should now proceed to the next stage, adding custom targets into Enterprise Manager.

Stage 12 – Add a custom target

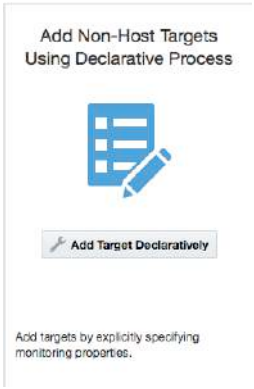
Once the plugin has been successfully deployed to the OMS and agent tiers, a custom target can be added into Enterprise Manager.

The following example illustrates the adding of a custom mongoDB target:

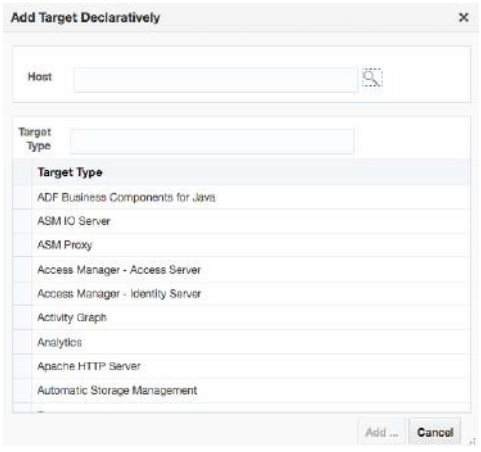
Setup->Add Target-> Add Targets Manually



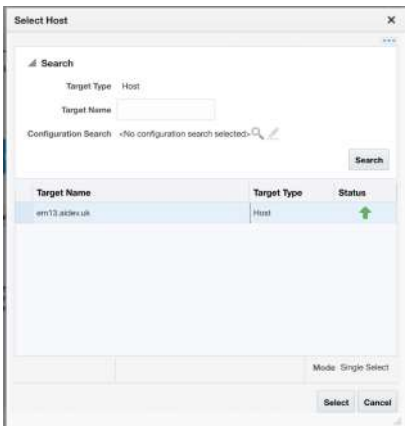
Choose ‘Add Non-Host Targets Using Declarative Process’:



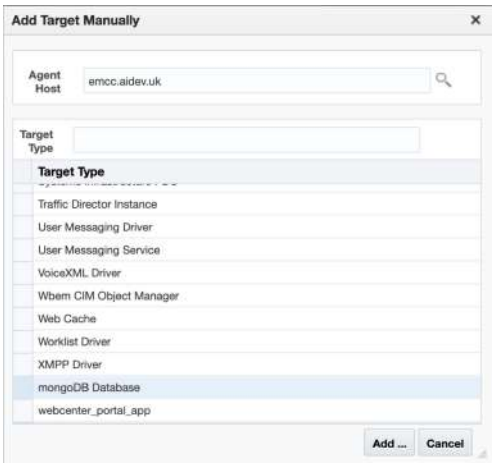
Click the spyglass:



Select the management agent host to add the target to, then click **Select**

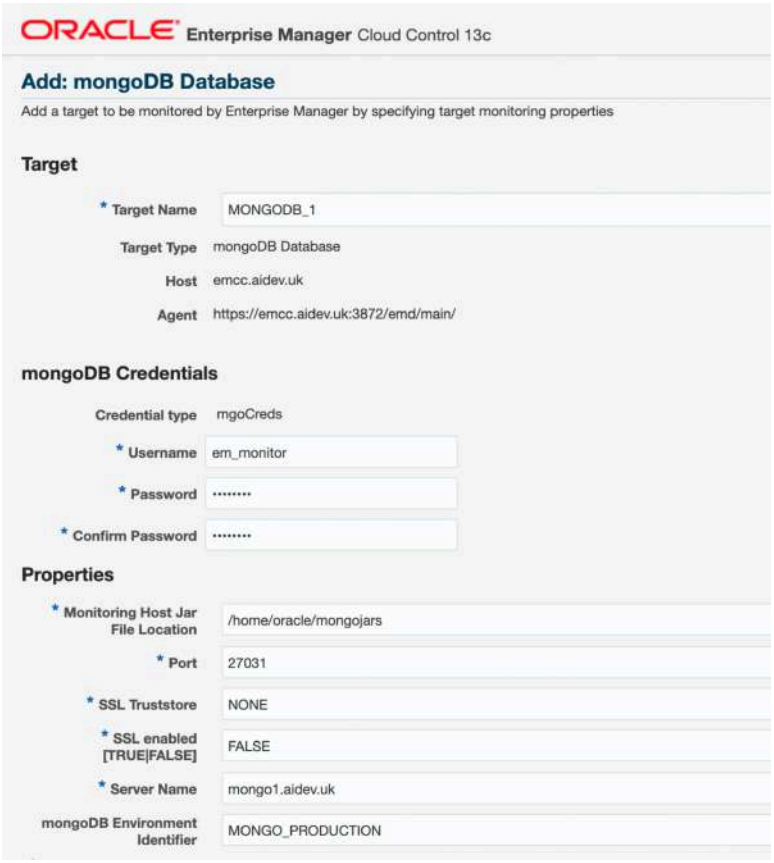


Select the custom target type in the target type box:



Click **Add** to view the Target Properties screen

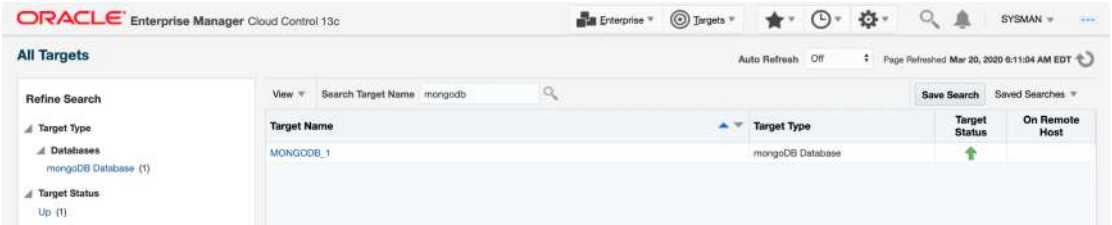
The target properties screen will be shown:



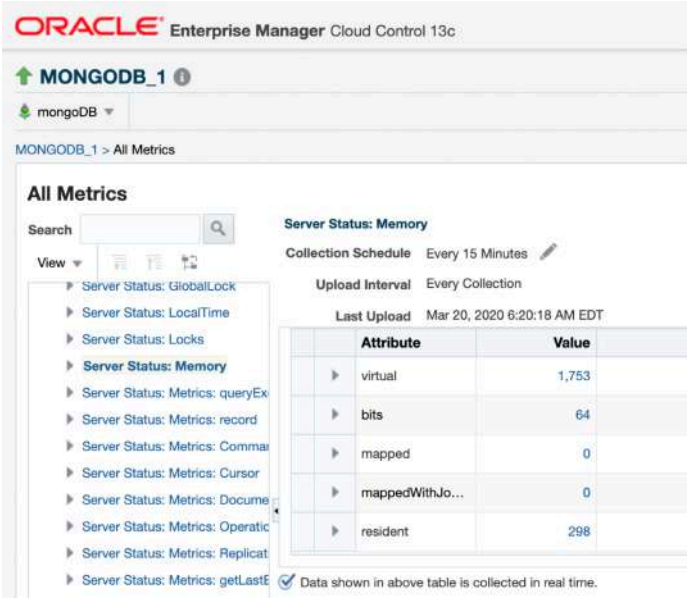
Complete the required properties and click OK

The target will be added into Enterprise Manager:

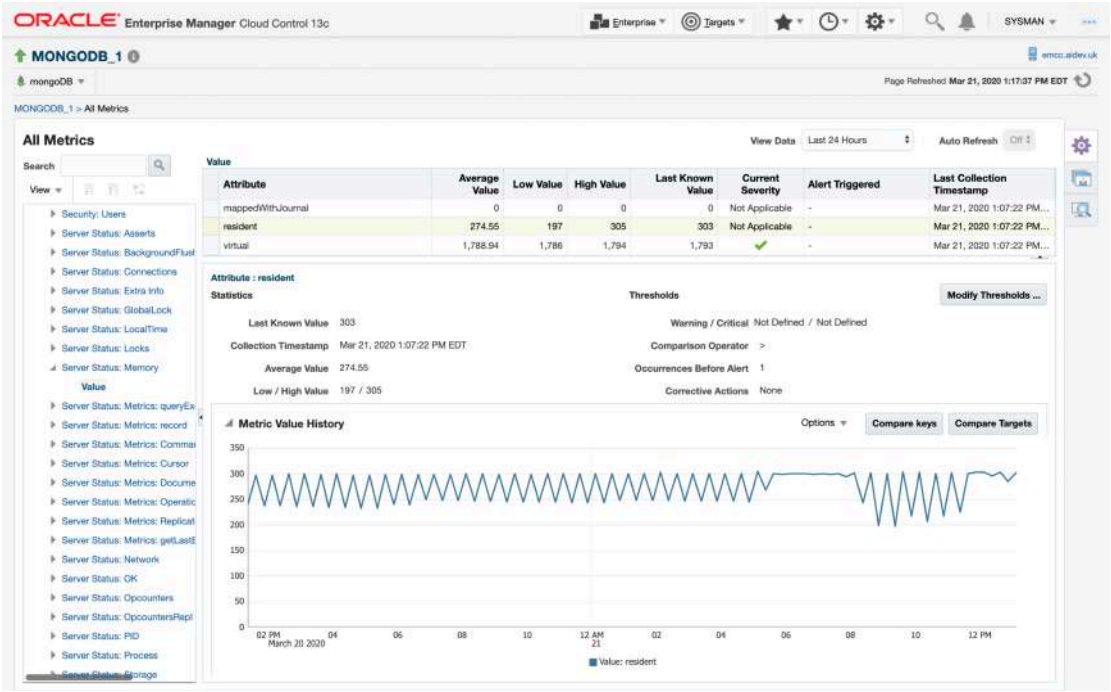
The target should now be visible and ‘Up’ in Enterprise Manager:



On the target home page, the ‘All Metrics’ link will show the metrics defined for the target type. The following example illustrates the mongoDB memory metric defined earlier in this document:

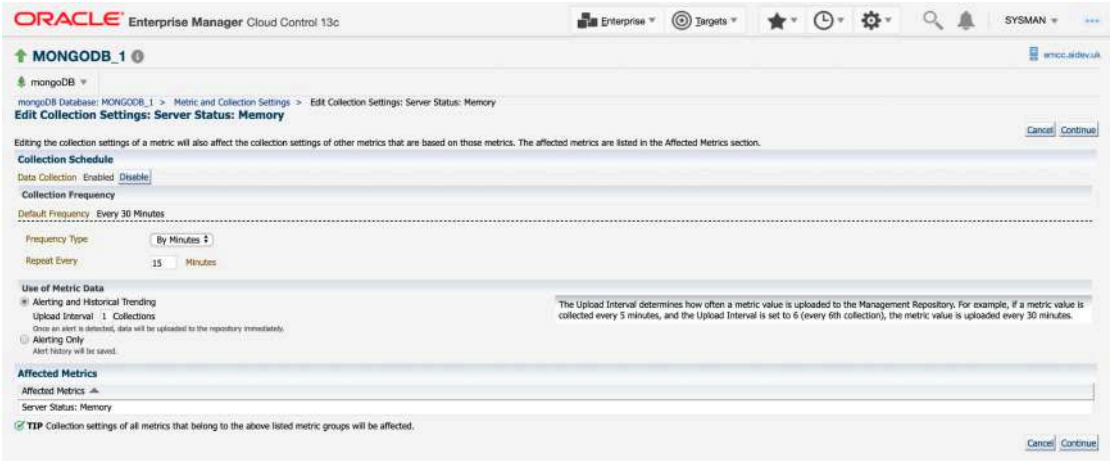


After a period of time, the user can examine historical metric information for the target. The following example shows historical data for the mongoDB memory metric:

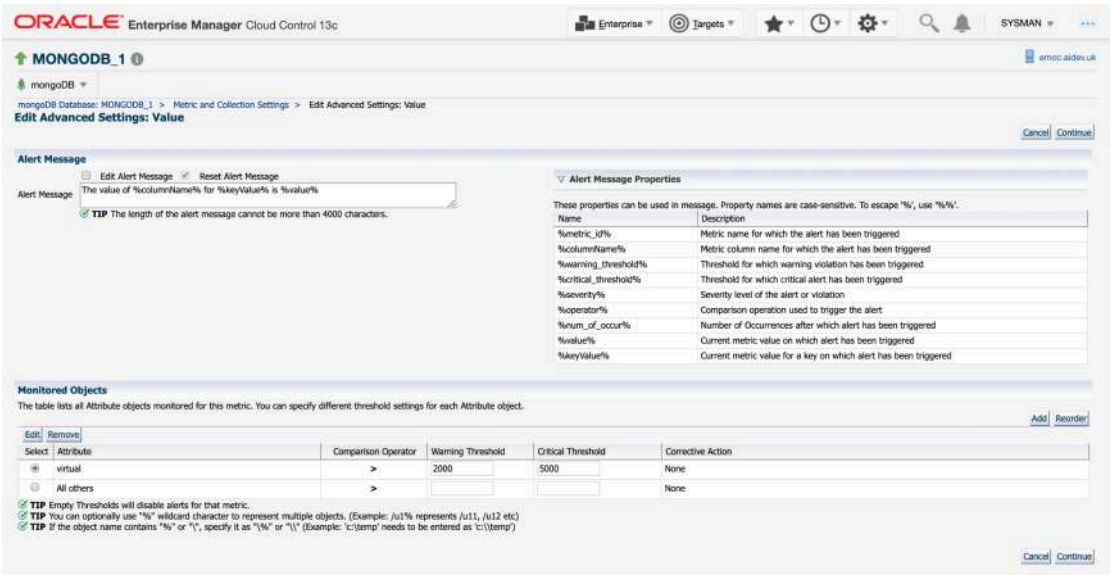


Metric collection behavior can be amended through the 'Metric and Collection Settings' link.

The following example illustrates the mongoDB plugin memory metric default collection settings:



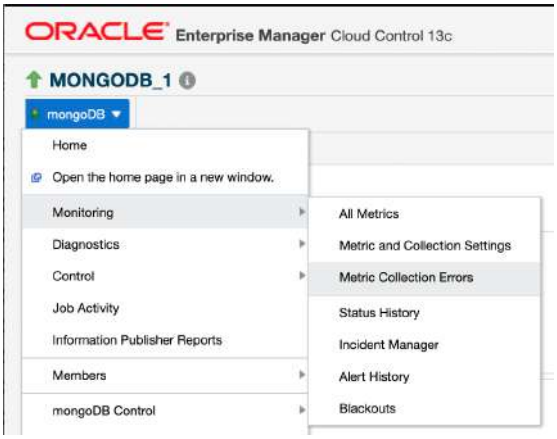
Custom key-based alerting thresholds can also be added. In this example, we have added Warning and Critical thresholds for the value obtained for the *virtual* key:



Once a target had been added, the plugin developer should test each metric from within Oracle Enterprise Manager.

Data collected and rendered within Enterprise Manager should be verified as being appropriate and correct.

The Metric and Collection Errors menu option can be used to identify issues:



Any issues or metric collection errors should be investigated, resolved, re-tested and the plugin source updated accordingly.

This action may involve additional tracing on the Enterprise Manager agent or by adding debug scripting into the backend code.

As the plugin code evolves, developers can use the '`emctl register oms metadata`' command to update Oracle Enterprise Manager with the changes made to core XML files.

Alternatively, if using a VM based development environment, Oracle Enterprise Manager can be quickly restored. Newly compiled opar files can then be imported, redeployed and subsequently tested again.

Following the successful undertaking of this stage, the plugin is nearing completion.

Developers should now progress to the next stage, custom UI development.

Stage 13 – Custom UI development

Once the plugin is imported into Enterprise Manager, and a target has been added, the plugin developer can create the custom plugin UI.

This is the final stage in the development process.

The plugin UI should be created using Oracle's JavaScript Extension Toolkit (JET) through the Apache Netbeans IDE.

Oracle has provided sample Netbeans projects within the EDK sample project code. We recommend using these as a starting point for UI development.

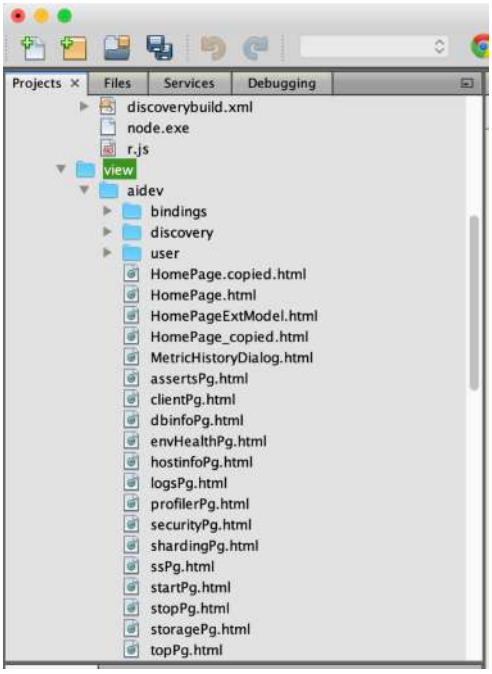
Apache Netbeans allows the plugin developer to create a JavaScript library that can be added into the plugin staging area and included within the final compiled opar file.

Once imported to the OMS, this library is used by Enterprise Manager to render the UI pages for target types contained within the custom plugin.

The Netbeans project contains code for each custom UI page – a JavaScript controller and a corresponding HTML based view page.

All controller code is contained within the js branch of the Netbeans project:

All screen content is defined in HTML files, within the View branch of the project:



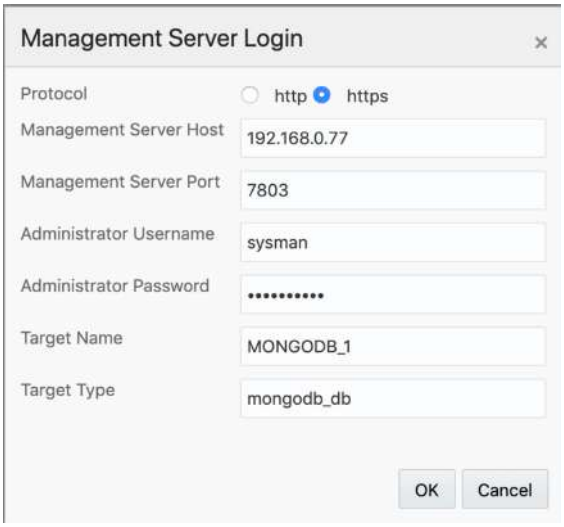
Netbeans allows the plugin developer to iteratively code and test in a standalone IDE, independent of Oracle Enterprise Manager.

Plugin interfaces can be run in a standard web browser locally, allowing quick development and testing of each UI page.

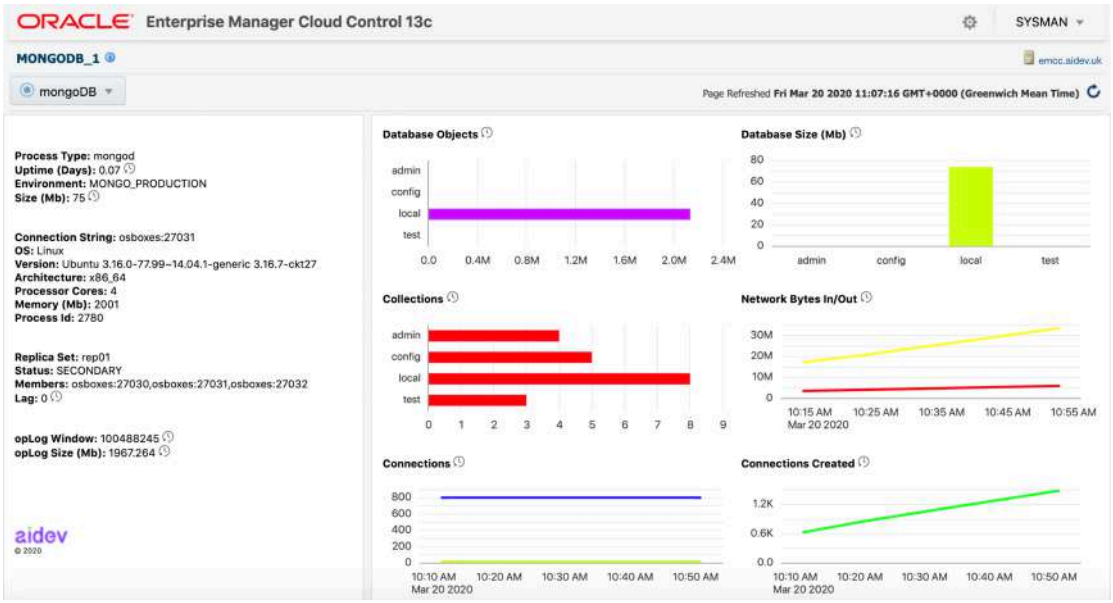
We typically develop UI screens using Netbeans and the Google Chrome browser.

Chrome also provides a Netbeans connector – this allows for deeper tracing and debugging of the UI from within Netbeans.

The logon screen within the Netbeans test harness allows for custom UI code to be run against a specific target in Enterprise Manager:

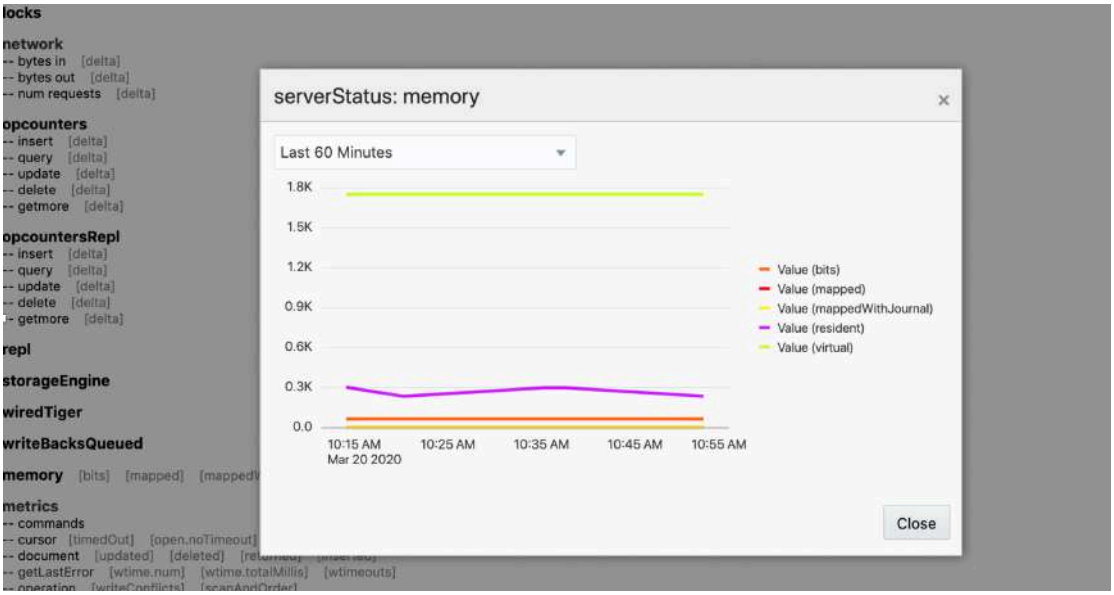


The following example is of the mongoDB target home page for the target added earlier in this document:



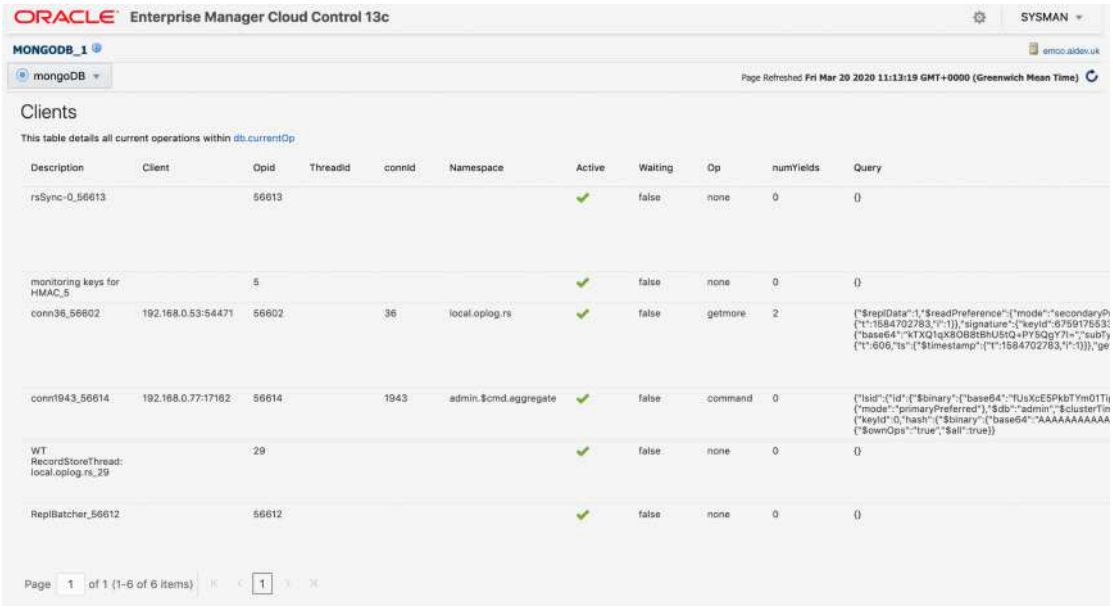
Metric information can be retrieved from Enterprise Manager and rendered graphically within the UI pages.

This screen illustrates the values obtained for the *memory* metric defined earlier in this document (all data on this screen is historical and retrieved directly from the Oracle Management Repository):

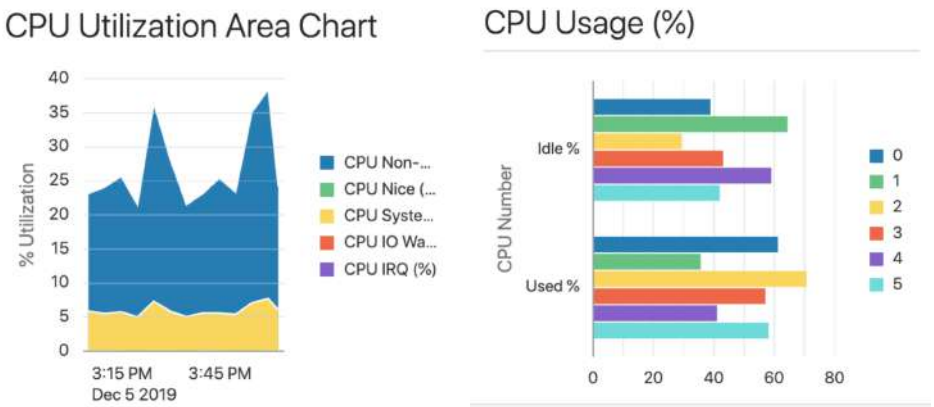


Screens can also capture and render real-time metric data, obtained directly from the custom target by the Enterprise Manager agent.

This example shows currently connected clients, captured real-time from mongoDB:



Traditional legacy Flex-type objects can be used within the UI pages – this is enabled through the inclusion of mpcui libraries within the supplied Netbeans project code.



Flex-type objects enable plugin developers with Flex experience to move to JET based coding with relative ease.

They also allow legacy Flex based plugins to be migrated to JET relatively quickly.

Oracle has documented the similarities between Flex and JET objects here.

For example, an availability data service in JET:

```
<mp-avail-data-service id="ads" params="targetName:appModel.target.name,
        targetType:appModel.target.type, days:1">
</mp-avail-data-service>
```

Compared to the Flex equivalent:

```
<mp:AvailDataService id="ads"  targetName="{appModel.target.name}"
                    targetType="{appModel.target.type}"/>
```

We recommend using Oracle's documentation as a starting point for UI development in JET.

The sample plugin project code contained in the EDK is also an excellent reference point for developers.

For the purpose of UI development testing, newly compiled JavaScript libraries can be imported directly into Oracle Enterprise Manager from a stage location:

```
emctl register oms metadata -service mpcui -file
/stage/mpcui/mongodb_database.xml -pluginId aidev.mongo.xdbs
```

This allows the developer to test UI changes from within Oracle Enterprise Manager.

Once UI development is complete, the resultant JavaScript libraries can be placed within the plugin code staging area.

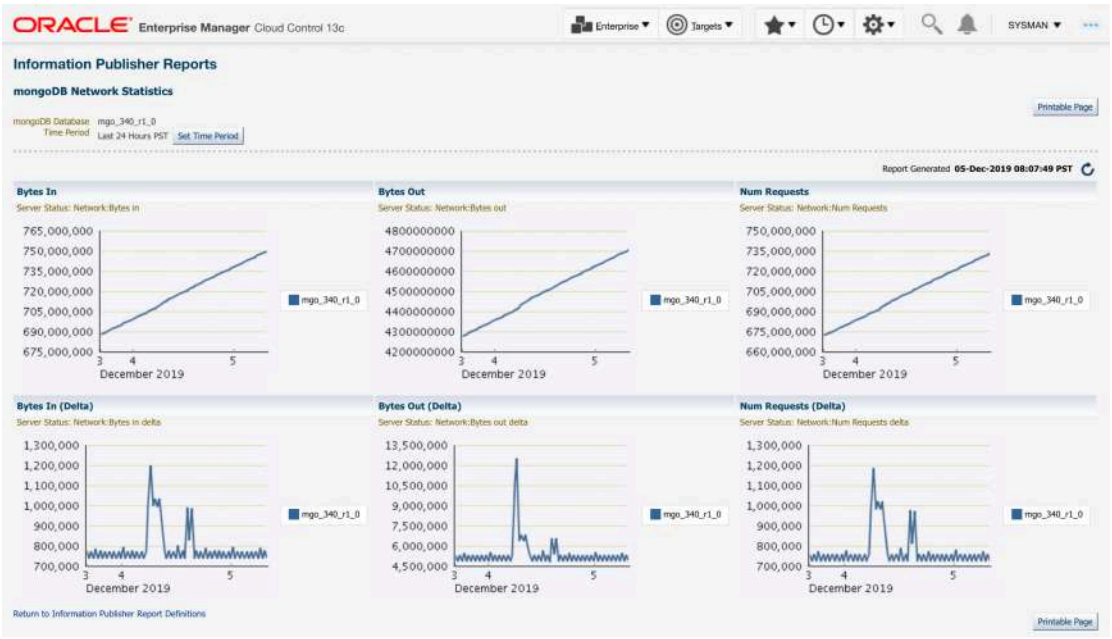
The finished plugin opar file can then be compiled and imported into a clean Enterprise Manager environment for final testing.

Additional Considerations

Outside the scope of this document, some other areas of consideration when developing a plugin are:

Custom Reports

Developers can include BI Publisher and Information Publisher reports in their plugin code. Reports can be easily developed within Enterprise Manager then copied to the plugin staging area before opar recompilation.



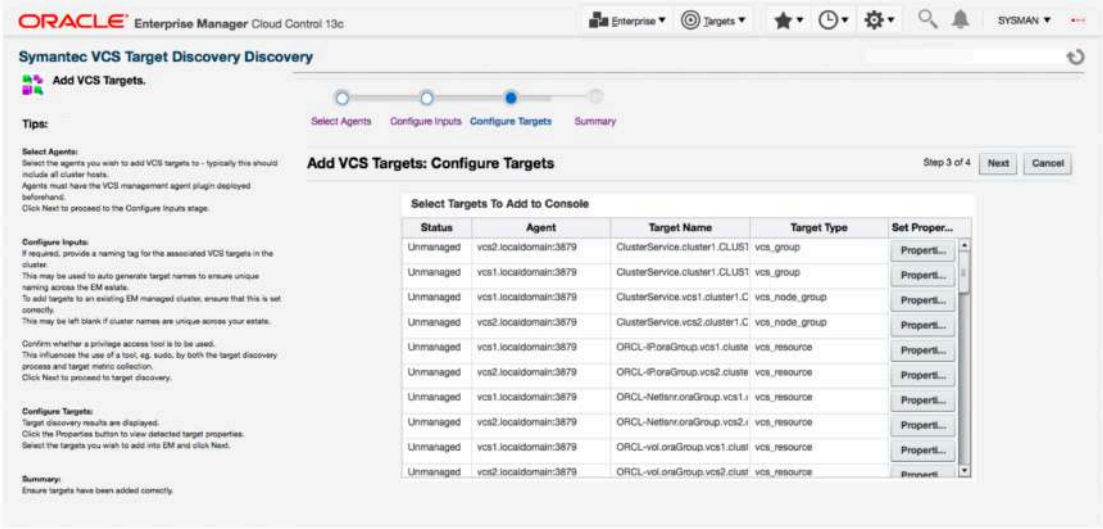
Custom Jobs

Job definitions can be contained within a plugin's content.

Our mongoDB plugin facilitates component control as well as providing the ability to remotely execute JavaScript code against targets. These operations can be run seamlessly within the Oracle Enterprise Manager jobs framework.

Target Discovery

Plugins can use guided target discovery. Our VCS plugin allows for target discovery, initiating custom code to identify VCS components on a host, passing the relevant target and association information back to Enterprise Manager.

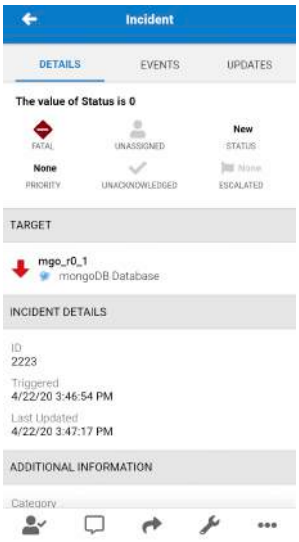


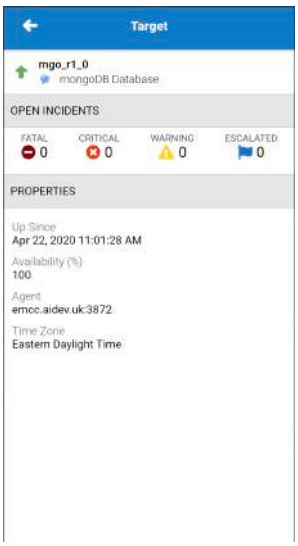
More information on the above features can be found within the Extensibility Programmer's Guide.

Oracle Enterprise Manager Mobile Application

Oracle Enterprise Manager 13.4 introduces a new mobile application – this allows the EM administrator to seamlessly interact with the EM monitoring and incident management framework through an Android/Apple app.

Plugins do not require any additional functionality to leverage this capability – integration is provided out of the box, allowing remote monitoring and incident management of custom targets through a mobile device.





Conclusion

At AIDEV we have found that the Oracle Enterprise Manager 13c EDK enables rapid and powerful development of custom plugins.

By following the stages outlined in this document, plugin developers can create enterprise-grade plugins to meet custom requirements.

Plugins created in this manner allow for a seamless integration into Oracle Enterprise Manager 13c, leveraging full use of core features such as monitoring, alerting, reporting and configuration management.

Apache Netbeans allows for iterative development of feature-rich JET based screens, rendering core target metric and configuration information to the end-user.

The inclusion of mpcui libraries within the EDK sample projects reduces the learning curve experienced by Flex developers and allows for a smooth migration of existing Flex-based plugins to a JET/js/HTML format.

Further Information

Aidev
http://www.aidev.uk

Oracle Enterprise Manager 13c
https://www.oracle.com/technetwork/oem/enterprise-manager/overview/index.html

Oracle Enterprise Manager Extensibility Exchange
https://apex.oracle.com/pls/apex/oracle_enterprise_manager/r/em-extensibility-exchange-v3

Oracle Enterprise Manager 13c Extensibility Documentation
https://docs.oracle.com/cd/cloud-control-13.3/nav/extensibility.htm

AIDEV/Wardrop Consulting Ltd is an Oracle Partner based in the UK, specializing in Oracle Enterprise Manager.

We offer Oracle Enterprise Manager consultancy services, including custom plugin development for application vendors.

Please contact info@aidev.uk for more information.