



EMC® Watch4net®

Version 6.3u2

Smarts Collector Administration Guide

P/N: pending

Rev : 41143

EMC²

Copyright ©2007 - 2014 EMC Corporation. All rights reserved. Published in the USA.

Published April, 2014

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

The information in this publication is provided as is. EMC Corporation makes no representations or warranties of any kind with respect to the information in this publication, and specifically disclaims implied warranties of merchantability or fitness for a particular purpose. Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

EMC², EMC, and the EMC logo are registered trademarks or trademarks of EMC Corporation in the United States and other countries. All other trademarks used herein are the property of their respective owners.

For the most up-to-date regulatory document for your product line, go to EMC Online Support (<https://support.emc.com>).

Contents

Preface	iii
Document Organization	iii
Where to get help	v
1 General configuration	1
1 Poller configuration file (smarts-*.xml)	2
2 Discovery configuration file (sm-discovery.xml)	5
2 Domain Descriptor	9
1 The data element	9
1.1 Basic usage	9
1.2 <i>EMC² Smarts InCharge</i> instance identifier	10
2 Exploring object relationship	11
2.1 Simple Relations	11
2.2 Polled properties scope	12
2.3 Qualified relations	12
2.4 Restricted relation sets	13
2.5 Special relations	14
2.6 Conditional Relationship Exploration	14
2.7 Contextual Relationship Exploration	15
3 Property polling and processing	16
3.1 Basic transformations	16
3.2 Multiple/Tabular properties handling	16
3.3 Fake variable generation	17
4 Value polling and processing	17
4.1 Basic transformations	17
4.2 Simple sanity checks	18
4.3 Inlined expressions	19
5 Embedded scripts	19
5.1 Script definition	19

5.2	Script reference	20
5.3	Script variables	21
5.4	General script context	21
5.5	Scripted values	24
5.6	Scripted properties	26
5.7	Instance filter scripts	27
A	Configuration hints	29
1	<i>EMC² Smarts InCharge</i> settings	29
1.1	Polling period	29
1.2	Interface duplex mode setting	30

Preface

Document Organization

1. General configuration	This section describes the <i>EMC² Smarts InCharge</i> collector general configuration.
2. Domain Descriptor	This section describes the domain descriptor configuration.

Where to get help

EMC support, product, and licensing information can be obtained as follows:

PRODUCT INFORMATION - For documentation, release notes, software updates, or information about EMC products, go to EMC Online Support at:

<https://support.emc.com>

TECHNICAL SUPPORT - Go to EMC Online Support and click Service Center. You will see several options for contacting EMC Technical Support. Note that to open a service request, you must have a valid support agreement. Contact your EMC sales representative for details about obtaining a valid support agreement or with questions about your account.

General configuration

The *EMC² Smarts InCharge* collector retrieves data from any domain to add historical capabilities to the suite. It is highly flexible and customizable to fit any situation.

To use this collector, simply declare it in the collecting configuration file (typically `<collector-manager-home>/conf/collecting.xml`) as shown in the following example:

```
<collector name="..." next="..." type="Sm-Collector" config="Smarts-Collector/Default/conf/smarts-pm.xml" />
```

If you are interested in polling multiple domains from the same *EMC² Smarts InCharge* broker, you might want to use the *EMC² Smarts InCharge* discovery collector instead. This collector connects to the broker and tries to create one or more suitable collectors for these domains.

To use the discovery collector, simply declare it in the collecting configuration file (typically `<collector-manager-home>/conf/collecting.xml`) as shown in the following example:

```
<collector name="..." next="..." type="Sm-Discovery" config="Smarts-Collector/Default/conf/sm-
discovery.xml" />
```

1 Poller configuration file (smarts-*.xml)

```
<config>
  <!-- ... -->
  <domain>INCHARGE-AM-PM</domain>
  <broker>localhost:426</broker>
  <username>admin</username>
  <password>changeme</password>
  <broker-username>admin</broker-username>
  <broker-password>changeme</broker-password>
  <!-- ... -->
</config>
```

You have to define one *EMC² Smarts InCharge* collector per domain (each one will have a different configuration file). In each configuration file, you will have to supply your *EMC² Smarts InCharge* credentials (username and password elements) for the given domain (domain element), on the given broker (broker element). If your broker requires authentication, you can add broker-username and broker-password elements to specify credentials. Please note that passwords can be encrypted by the APG password encryption tool.

All the other settings are related to the polling process. You can choose your polling period in seconds (period element) according to your domain configuration and your needs. The collector will assign the string specified in the group element to generated raw values. This group should match one of your processing component configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "smarts.dtd">
<config>
  <!-- ... -->
  <group>group</group>
  <period>240</period>
  <!-- ... -->
</config>
```

The *EMC² Smarts InCharge* Collector is able to retrieve numerical values from domains but also some object properties to tag variables with. It is a very important feature since these properties will enable you to classify data in the portal using filtering. Property polling is controlled via the `properties` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "smarts.dtd">
<config>
  <!-- ... -->
  <properties refresh="00:00/86400" send-on-refresh-only="true" />
  <!-- ... -->
</config>
```

The `refresh` attribute controls how often the collector should set the `refresh` tag on the generated values. It can be either a time with a period (00:00/86400 will refresh properties each day at midnight) or a number of polling loop (360 will refresh properties every 360 polling loops). In both cases, properties will always be refresh on first polling loop, right after the collector startup.

The `send-on-refresh-only` attribute controls the way the collecting component will send properties to the processing component. If set to true, it will send properties only when they are refreshed. Else, it will always send them. This behavior will affect the way *APG* discovers new indicators. Indeed, since the collector is stateless, it cannot figure out which indicator is new, and which one is already existing. Then, it assumes that every indicator may be new and then always send properties along with values (in order to create a tagged value which will be classified immediately in the portal).

The few other settings a related to polling performance tuning. The *EMC² Smarts InCharge* collector is designed to be extremely scalable and then should be able to poll huge domains. To get the best out of it, there are several parameters to tune.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "smarts.dtd">
<config>
  <!-- ... -->
  <thread pool-size="20" />
  <dm-connection pool-size="5" keep-connection="false" creation-grace-time="100" />
  <smooth-factor>0.5</smooth-factor>
  <!-- ... -->
</config>
```

The most important ones are related to the parallelism. They are controlled by the `thread` and `dm-connection` elements.

The *EMC² Smarts InCharge* Collector is an high performance, asynchronous process, which gather data from *EMC² Smarts InCharge* in-memory repository very efficiently. The `thread pool-size` attribute let you specify the number of concurrent pooling threads which will share a customizable number of connections to *EMC² Smarts InCharge* domain (specified in `dm-connection pool-size`). Since the Collector is design to be very fast, it can disturb *EMC² Smarts InCharge* domain: that's why there are two other parameters which control the pooling rate. The `dm-connection creation-grace-time` is the minimum delay between consecutive connection to the *EMC² Smarts InCharge* domain. The `smooth-factor` control the actual pooling rate. For example, if pooling period is 240 second and smooth factor is 0.5, the Collector will try to pool the domain in $0.5 * 240 = 120$ seconds.

The `dm-connection keep-connection` attribute control whether we should keep domain connections after the polling process or not. It is usually more reliable to set it to `false` ensuring that we always have proper connectivity to the domain manager.

Finally, the most important configuration element of a *EMC² Smarts InCharge* Collector is its domain descriptor file which defines what is interesting in a *EMC² Smarts InCharge* domain. You specify it in the `indicators` element and the syntax of this file is described in the following sections.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "smarts.dtd">
<config>
  <!-- ... -->
  <indicators select="both">conf/pm.xml</indicators>
  <!-- ... -->
</config>
```

The `select` attribute enable you to choose between 3 polling modes:

- `properties` will poll only properties, not generating any actual performance metrics. This is useful for inventory purpose.
- `values` will generate only the performance values and not poll properties at all.
- `both` is the default polling mode, polling both properties and values.

The complete configuration file should look like that:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "smarts.dtd">
<config>
  <domain>INCHARGE-AM-PM</domain>
  <broker>localhost:426</broker>
  <username>admin</username>
```

```

<password>changeme</password>
<broker-username>admin</broker-username>
<broker-password>changeme</broker-password>
<group>group</group>
<period>240</period>
<thread pool-size="20" />
<dm-connection pool-size="5" keep-connection="false" creation-grace-time="100" />
<indicators select="both">conf/pm.xml</indicators>
<properties refresh="00:00/86400" send-on-refresh-only="true" />
<smooth-factor>0.5</smooth-factor>
</config>

```

2 Discovery configuration file (sm-discovery.xml)

```

<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.watch4net.com/APG/Smarts/DomainDiscovery1">
  <authentication>
    <broker>localhost:426</broker>
    <broker-username>admin</broker-username>
    <broker-password>changeme</broker-password>
    <domains refresh-period="240">
      <default-username>admin</default-username>
      <default-password>changeme1</default-password>
      <overrides>
        <domain name="INCHARGE-SA">
          <username>admin</username>
          <password>changeme2</password>
        </domain>
      </overrides>
    </domains>
  </authentication>
  <!-- ... -->
</config>

```

The discovery collector needs to be able to connect to the *EMC² Smarts InCharge* broker and every domain you want to get data from. The discovery process occurs every 4 minutes and is configurable with the `refresh-period` attribute. In the authentication element, you have to give the broker address (`broker`) and optionally, the broker credentials if broker authentication is enabled (`broker-username` and `broker-password`). By default, the discovery collector uses `default-username` and

default-password to connect to the discovered domains, however, if some domains do not have the same credentials, it is possible to specify them using the overrides. You can declare each exception here, like *INCHARGE-SA* in the previous example.

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.watch4net.com/APG/Smarts/DomainDiscovery1">
  <authentication>
    <!-- ... -->
  </authentication>
  <!-- first line to match is applied -->
  <domains default-exclude="true" ignore-case="true">
    <include>DOMAIN-CLIENT-TEST</include>
    <exclude>DOMAIN-*TEST</exclude>
    <include>DOMAIN-*</include>
  </domains>
  <templates>
    <!-- ... -->
  </templates>
</config>
```

Instead of processing all discovered domains, it is possible to restrict which domains to consider using the configuration element `domains`. In this previous example, only domains with a name starting with `DOMAIN-` are added except for the ones ending with `-TEST`. `DOMAIN-CLIENT-TEST` is also added. If set to true, `default-exclude` attribute will exclude domains not matching any rule. Otherwise, domains not matching any rules will be included. If set to true, the `ignore-case` attribute will ignore the case of rules expressions. Setting it to false will check for the case.

Here are the reserved characters:

- `*` will match any number of characters
- `?` will match only one character
- `\` is used to escape reserved characters (`\\` `\*` `\?`)

Note: The first line to match will apply. Next lines will not be tested.

To know which *EMC² Smarts InCharge* collector the discovery process should create for each discovered domain, it passes through some `templates`. A template consists of some criteria (features)

and some collector configurations (configuration). If the domain matches the criteria, all the corresponding collector configurations will be used and no more templates will be evaluated for this domain. If none of the templates match, the domain uses the default template.

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.watch4net.com/APG/Smarts/DomainDiscovery1">
  <!-- ... -->
  <templates>
    <template>
      <!-- AM/PM Domains -->
      <features>
        <feature>Feature-DEVSTAT</feature>
        <feature>Feature-PERFORMANCE</feature>
        <feature>ICIM_Manager::ICIM-Manager::AMPM_Mode=
AVAILABILITY_AND_PERFORMANCE_MANAGER</feature>
      </features>
      <configuration>conf/smarts-pm-health.xml</configuration>
      <configuration>conf/smarts-pm.xml</configuration>
    </template>
    <template>
      <!-- PM Domains -->
      <features>
        <feature>Feature-DEVSTAT</feature>
        <feature>Feature-PERFORMANCE</feature>
        <feature>ICIM_Manager::ICIM-Manager::AMPM_Mode=PERFORMANCE_MANAGER</feature>
      </features>
      <configuration>conf/smarts-pm-health.xml</configuration>
      <configuration>conf/smarts-pm.xml</configuration>
    </template>
    <template>
      <!-- ACM Domains -->
      <features>
        <feature>Feature-AI_ACM</feature>
      </features>
      <configuration>conf/smarts-pm-health.xml</configuration>
      <configuration>conf/smarts-acm2.xml</configuration>
    </template>
    <default>
      <!-- Other Domains -->
      <configuration>conf/smarts-pm-health.xml</configuration>
    </default>
  </templates>
```

```
</config>
```

There are two types of tests you can do on a domain to determine its type:

- Test the presence of a specific `InCharge_Feature` instance. To do this, simply use a `feature` element that contains the name of the instance you are looking for. For example, `<feature>Feature-PERFORMANCE</feature>` checks if the object `InCharge_Feature::Feature-PERFORMANCE` exists.
- Test the value of a specific object field. To do this, include the field to test in a `feature` element using the syntax `class::instance::field=value`. For example, `<feature> ICIM_Manager::ICIM-Manager::AMPM_Mode=PERFORMANCE_MANAGER</feature>` makes sure that the `AMPM_Mode` field value of the object `ICIM_Manager::ICIM-Manager` is `PERFORMANCE_MANAGER`.

2

Domain Descriptor

The domain descriptor is an *XML* file which describes what will be retrieved from a specific *EMC² Smarts InCharge* domain.

1 The data element

For each class you are interested in, you have to create a data block. The `class` attribute contains the names of the interesting *EMC² Smarts InCharge* classes as a space separated list.

1.1 Basic usage

The nested `value` and `property` elements specify which numerical attributes and which object properties should be retrieved. Both `value` and `property` elements can have a `rename` attribute to rename the *EMC² Smarts InCharge* attribute. The `value` supports a special attribute called `unit` which enables you to specify the value unit (you may find it useful to present data in the portal).

Here is a very simple example.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface" start="true">
    <property name="Name" rename="part"/>
    <value name="CurrentUtilization" rename="Utilization" unit="%" />
  </data>
  <!-- ... -->
</domain>
```

This little example will collect the *CurrentUtilization* value from all instances of *Interface*. The value will be called *Utilization* in APG (using the *name* property) since it has been renamed. Each value will be tagged with the *DisplayName* attribute (renamed to *part*) of the corresponding *EMC² Smarts InCharge* object and with a *unit* property containing %.

Please note the *start* attribute. It is *VERY* important not to forget it. It is used by the collector as an "entry point" into *EMC² Smarts InCharge* object repository. When the collector starts the polling process, it will retrieve all the instances of the classes whose *start* attribute is set to *true*.

1.2 EMC² Smarts InCharge instance identifier

By default, the *EMC² Smarts InCharge* instance name is used by APG as a globally unique key for the generated values. However, the *EMC² Smarts InCharge* instance name may change across re-discoveries. While it has absolutely no impact on *EMC² Smarts InCharge* functionality, this will make APG think that a new device has been discovered.

The *instance-name-pattern* attribute let you change this behaviour. You can construct a new instance identifier using class attributes. The general syntax is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface" instance-name-pattern="IF-{SystemName}-{DeviceID}">
    <!-- ... -->
  </data>
  <!-- ... -->
</domain>
```

To create the new key, you simply have to create a pattern where names between { and } signs are class attributes. The previous example would generate instance names like:

- IF-10.128.3.1-eth0
- IF-10.128.7.1-eth0
- ...

2 Exploring object relationship

EMC² Smarts InCharge stores the topology using many objects linked together through relations. The collector can use these relations to explore the topology. It enables you to collect properties and values from many linked objects from the instances of a given class. You can do this by specifying a space separated list of relation fields in the `containment` attribute of the `data` element. Relation fields can be either single reference or reference sets.

2.1 Simple Relations

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface" containment="InstrumentedBy" start="true">
    <!-- ... -->
  </data>
  <data class="NetworkAdapter_Performance">
    <!-- ... -->
  </data>
  <!-- ... -->
</domain>
```

In this example, the collector now explores the *InstrumentedBy* relation of the *Interface* instances looking for a known class (i.e. declared in a data block without `start` attribute or with `start="false"`). Then, if the *InstrumentedBy* contains some *NetworkAdapter Performance* instances, the collector will retrieve the *ifInOctetsRate* attribute from the *EMC² Smarts InCharge* object. The generated value will keep any previously assigned properties. Therefore, the *ifInOctets* values will also be tagged with the interface name retrieved from the *Interface* instance. You will find more detail on this in the next section.

Please note that the `start` attribute is missing (*false* by default). This means that the collector will not try to retrieve every instances of the class *NetworkAdapter Performance* but will know what to do if it encounters one navigating through a relation.

2.2 Polled properties scope

By default, polled properties are visible within their own data block and every other block we may reach from it. A data block can override a property and then shadow any previous value. When the collector exits a data block, it will normally discard the collected properties.

It is sometimes useful to retrieve properties from other instances using relationships, and tag local numerical attributes with them. That is why you can set the `keep-properties` attribute to *true* in order to extends the property lifetime to the previously explored data block.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface" containment="Underlying_InstrumentedBy" start="true">
    <!-- ... -->
  </data>
  <data class="NetworkAdapter.Performance">
    <value name="ifInOctetsRate" rename="ifInOctets" unit="Octets/s"/>
  </data>
  <data class="IP" keep-properties="true">
    <property name="Address" rename="ifaceip" />
  </data>
  <!-- ... -->
</domain>
```

In this example, we are exploring the *Underlying* relationship to get the interface IP address and tag *ifInOctets* values with.

2.3 Qualified relations

You can qualify your relation to make them easier to read but also to give hints to the poller on how this relation should be used. Relation attributes are written between parenthesis:

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

```
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface" containment="InstrumentedBy(to-perf-class)">
    <!-- ... -->
  </data>
  <!-- ... -->
</domain>
```

If the relation attributes contains the string `no-values` this will give a hint to the collector not to explore this relation when property polling is disabled. This is useful when some classes are explored only to gather properties.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface" containment="Underlying(ip-no-values)">
    <!-- ... -->
  </data>
  <!-- ... -->
</domain>
```

2.4 Restricted relation sets

In *EMC² Smarts InCharge* a relation can be either a single reference, or a reference set. If we want to process a reference set as a single reference, we can restrict the collector to retrieve only the first instance. This is achieved by adding a `[0]` at the end of the relation attribute name.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Host" containment="ComposedOf [0]" start="true">
    <!-- ... -->
  </data>
  <!-- ... -->
</domain>
```

In the example, we would only retrieve the first component from each *Host* instance. Please note that the [0] suffix *is not part* of a qualified relation name. We you will reference this relation, you should omit it.

2.5 Special relations

APG adds a special relation which doesn't exist in *EMC² Smarts InCharge*. This relation is called *self* and enables you to re-explore a class instance within another context.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface" containment="self(to-another-context)">
    <!-- ... -->
  </data>
  <!-- ... -->
</domain>
```

2.6 Conditional Relationship Exploration

Sometimes you don't want to explore all the relations anytime, or don't want to get all the instances from a relation. That why the *from* attribute has been introduced to the *data* element. With this attribute, you can choose in what context you want to explore a class instance.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data id="Device" class="Host" containment="ComposedOf" start="true">
    <!-- ... -->
  </data>
  <data id="Device" class="Router" containment="ComposedOf" start="true">
    <!-- ... -->
  </data>
  <data class="Processor" containment="InstrumentedBy" from="Host">
    <!-- ... -->
  </data>
  <data class="Interface" containment="InstrumentedBy" from="Device">
```

```

<!-- ... -->
</data>
<!-- ... -->
</domain>

```

The `from` attribute can contain a space separated list of data block class names or data block ids. In the previous example, we would explore *Processor* block from *Hosts* only while we would explore *Interfaces* for every *Device*. Please note that the class names specified in the `from` attribute **WILL NOT HONOR CLASS HIERARCHY**. It must be the exact name specified in the `class` attribute of the data block the collector comes from.

When several choices are involved, the following rules are applied:

1. The collector firsts locates matching data block (with or without `from` attribute).
2. If blocks with matching `id` are found, we explore them.
3. If there are no block with matching `id` but some with matching class names, we explore them.
4. If no matching blocks were found in 2 and 3, we explore default matches (without `from` attribute).

2.7 Contextual Relationship Exploration

The `from` attribute can contain even more specific references. The `id` or `class` name specified can be appended a qualified relation name (within brackets). The data block will be evaluated according the rules described in the previous section, but only if the collector is coming from the specified relation. This is particularly useful with the `self` relation to avoid infinite recursion.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface" containment="self(special-context)">
    <!-- ... -->
  </data>
  <data class="Interface" from="Host[self(special-context)]">
    <!-- ... -->
  </data>
  <!-- ... -->
</domain>

```

3 Property polling and processing

3.1 Basic transformations

A transformation element nested into a property one performs a simple search and replace in the collected attribute. You have to specify two attributes `match` and `replace` which are respectively a regular expression and the new content to replace the match with. They are always all orderly evaluated and then a value can be transformed several times.

The property element also support a `drop-empty` attribute which default value is `true`, meaning that empty properties will be discarded.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface">
    <property name="MaxSpeed" rename="maxspeed" drop-empty="true">
      <transformation match="^([0-9]{2,})[0-9]{6}$" replace="$1 Mbits"/>
      <transformation match="^([0-9]{2,})[0-9]{3}$" replace="$1 Kbits"/>
    </property>
  </data>
  <!-- ... -->
</domain>
```

Here, we collect the *MaxSpeed* attribute from *Interface* instances transforming it into a more human readable string. Any empty speed value will be discarded.

3.2 Multiple/Tabular properties handling

The default behavior of property polling is to always keep the first value for each distinct property name within a data block. However, if multiple values should be concatenated, you can specify the `append-with` attribute. Instead of being discarded, additional values will be appended to the previous one using the content of the `append-with` as a separator.

Please note this also applies with the `keep-properties` set to `true`, when already existing properties are retrieved from relation.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
```

```

<!-- ... -->
<data class="HostSystem">
  <property name="Vendor_OperatingSystem_OSLevel"
    rename="devdesc.devdesc.devdesc" append-with=" " />
</data>
<!-- ... -->
</domain>

```

In this example, we will append Vendor, OperatingSystem and OSLevel with a `<space>` character to generate a nice devdesc property.

3.3 Fake variable generation

When reports requires a lot of properties, there is a specific feature to reduce the number of non-fundamental ones. Indeed, sometimes you need some properties only for report decoration and never for filtering. This kind of properties doesn't have to be tagged on every variable but only one. For such properties, you have to set the attribute `make-variable` to `true`.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface" start="true">
    <value name="CurrentUtilization" rename="Utilization" />
    <property name="Name" rename="part"/>
    <property name="Type" rename="iftype" make-variable="true"/>
  </data>
  <!-- ... -->
</domain>

```

In this example, the collector will create an initialized variable (non numerical values) with the `iftype` property set. Note that values *Utilization* will never be tagged with the *iftype* property.

4 Value polling and processing

4.1 Basic transformations

A transformation element nested into a value one performs a string to number conversion. The `match` attribute is a regular expression and the `replace` attributes the number that should be used if the string matches. If the string doesn't match any transformation `match` regular expression, it will be

discarded. If the value matches one regular expression, transformation process will stop there.

If the value tag specifies the `force` attribute, instead of being discarded, an invalid value will be generated as *uninitialized*. This is useful to get the metric into *APG* even when there is no actual value to send.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="Interface" containment="InstrumentedBy">
    <value name="Status" rename="Availability" unit="%" force="true">
      <transformation match="UP|up" replace="100.0"/>
      <transformation match="DOWN|down" replace="0.0"/>
    </value>
  </data>
  <!-- ... -->
</domain>
```

Here, we make use of the *Status* attribute to generate an availability value. The string value is converted from UP to 100 and from DOWN to 0. Since we have specified the `force` tag, any other value will result into an uninitialized value. The direct effect is that this interface will be shown in *APG* even if the status is unknown.

4.2 Simple sanity checks

The value elements support two attributes `min` and `max` which provide simple sanity check. If a value is outside specified boundaries, it will be discarded (unless the `force` attribute has been specified).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="NetworkAdapter_Performance">
    <value name="ifInOctetsRate" unit="Octets/s" min="0"/>
  </data>
  <!-- ... -->
</domain>
```

This example would drop negative (and obviously invalid) rates. Since we don't specify `max` attribute, any positive value will be accepted.

4.3 Inlined expressions

When specifying the source attribute for a value using the name attribute (e.g. name = "ifInOctetRate"), you can also write some simple expressions. Most arithmetic operators are supported (+, -, *, /). Please note that you can't use `<space>` in an expression since it would be splitted in two values: you must quote it first.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="HostPhysicalDevice">
    <value name="'NoOfBlocks*BlockSize'" rename="Capacity" unit="Bytes"/>
  </data>
  <!-- ... -->
</domain>
```

Since the capacity in bytes is not available here, we multiply the number of available block by the block size to compute it.

5 Embedded scripts

Sometimes the previously described features are not flexible enough to poll a *EMC² Smarts InCharge* domain. Some models can be very complex, requiring some advanced processing. This processing can be achieved embedding processing scripts in a domain descriptor.

5.1 Script definition

Scripts are declared in separate script elements. For each script you have to declare its id, its parameters (in) and its output (out). The script body is declared in the script elements itself.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <script id="my-script" in="Param1 Param2" out="Output1">
    <![CDATA[
      <!-- script body -->
    ]]>
  </script>
```

```
<!-- ... -->
</domain>
```

Scripting engine is based on *Apache JEXL*. The syntax is mostly *Java* based with some handy shortcuts but some restrictions. For more information on the exact syntax, please refer to <http://jakarta.apache.org/commons/jexl/reference/syntax.html>.

5.2 Script reference

A script can be referenced in a data block:

- to process values using a `scripted-value` element (detailed in section 5.5)
- to process properties using a `scripted-property` element (detailed in section 5.6)
- to filter instances using `pre-filter-script` element and/or `post-filter-script` element (detailed in section 5.7)

Each of these elements must specify the script reference using the `script-id` attribute. They must also specify input using the `in` attribute (except for `pre-filter-script`) and output using the `out` attribute (for `scripted-value` and `scripted-property` only).

Input is a list of space separated *EMC² Smarts InCharge* attribute names while output is a list of *APG* names. They are similar to the `name` and `rename` attributes in `value` and `property` elements. If they are not or partially specified, those declared in script definition will be used.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <data class="...">
    <!-- ... -->
    <scripted-value script-id="my-script" in="MyInAttribute" out="MyVar"/>
    <!-- ... -->
  </data>
  <!-- ... -->
</domain>
```

5.3 Script variables

Each script is evaluated in its own context. As in any programming language, variables can be referenced using identifiers. When a script is evaluated, each input parameter is defined as a read-only variable (in can be shadowed by a local variable however).

Input parameters are either *EMC² Smarts InCharge* repository objects directly gathered from domain using the *Remote API* (see reference for details) or constants (when enclosed by quotes in the input parameter list). A script will be exposed all attributes collected from the domain, not only those requested as input parameters.

A script can obviously use its own variables. Furthermore, it can access some special variables:

- `ctx` is the script context. Most functionality is based on this special variable. Depending on its purpose, a script `ctx` variable will expose different functionality.
- `smInstance` is the instance name from *EMC² Smarts InCharge*.
- `apgInstance` is the instance name from *APG* (it may be equal to `smInstance`).
- `pctx` is the polling context. It enables a script to interact with poller internals. You should never use it unless you know what you are doing.

5.4 General script context

The script context offer some helper methods for scripts. In any script you can access the following methods from the `ctx` variable:

<code>Object value(value)</code>	Unpacks a <i>Remote API</i> object.
<code>Object[] array(value)</code>	Unpacks a <i>Remote API</i> array.
<code>String str(value)</code>	Creates a string from the specified object, the returned string is never null but can be empty.
<code>String[] strarray(value)</code>	Creates a string array from the specified object, the returned string is never null but can contain empty strings.

<code>String nstr(value)</code>	Creates a string from the specified object, the returned string can be null.
<code>String[] nstrarray(value)</code>	Creates a string array from the specified object, the returned string can't be null but can contain null values.
<code>Number num(value)</code>	Creates a number from the specified object, the returned number can't be null but can be <i>NaN</i> .
<code>Number[] numarray(value)</code>	Creates a number array from the specified object, the returned number can't be null but can contain some <i>NaN</i> .
<code>Number nnum(value)</code>	Creates a number from the specified object, the returned number can be null.
<code>Number[] nnumarray(value)</code>	Creates a number array from the specified object, the returned number can't be null but can contain null values.
<code>void id(instance, key, suffix)</code>	Creates a unique id from the specified names (instance and suffix can be null).
<code>void store(pctx, id, value)</code>	Stores a value in the polling context using the specified unique id.
<code>Object fetch(pctx, id)</code>	Retrieve a value from the polling context using the specified unique id.
<code>Object timestored(pctx, id)</code>	Retrieves the timestamp when a value was stored from the polling context using the specified unique id.
<code>Object looptime-stored(pctx, id)</code>	Retrieves the polling cycle timestamp when a value was stored from the polling context using the specified unique id.

<code>void store(key, value)</code>	Stores a value in the current polling context using the specified key. Keys will use input parameter mappings.
<code>Object fetch(key)</code>	Retrieves a value from the current polling context using the specified key.
<code>Object timestored(key)</code>	Retrieves the timestamp when a value was stored from the current polling context using the specified key.
<code>Object looptime-stored(key)</code>	Retrieves the polling cycle timestamp when a value was stored from the current polling context using the specified key.
<code>void pstore(key, value)</code>	Stores a value in the previous polling context using the specified key. Keys will use input parameter mappings.
<code>Object pfetch(key)</code>	Retrieves a value from the previous polling context using the specified key.
<code>Object ptimestored(key)</code>	Retrieves the timestamp when a value was stored from the previous polling context using the specified key.
<code>Object plooptime-stored(key)</code>	Retrieves the polling cycle timestamp when a value was stored from the previous polling context using the specified key.
<code>void istore(key, value)</code>	Stores a value in the initial polling context (which is kept across polling loops) using the specified key. Keys will use input parameter mappings.
<code>Object ifetch(key)</code>	Retrieves a value from the initial polling context (which is kept across polling loops) using the specified key.
<code>Object itimestored(key)</code>	Retrieves the timestamp when a value was stored from the initial polling context (which is kept across polling loops) using the specified key.

<code>Object illoptime-stored(key)</code>	Retrieves the polling cycle timestamp when a value was stored from the initial polling context (which is kept accross polling loops) using the specified key.
<code>void out(value)</code>	Prints the value on standard output.
<code>void err(value)</code>	Prints the value on error output.
<code>void log(value)</code>	Prints the value in logs.
<code>int time()</code>	Returns the current timestamp in seconds.
<code>int looptime()</code>	Returns the current polling cycle timestamp in seconds.

5.5 Scripted values

Scripted values are scripts which generate values. Values are generated with the `create` methods. Every created value will be automatically sent without any other operation unless discarded with the `discard` method. These scripts can access the following methods from `ctx`:

<code>Value create(name)</code>	Creates an uninitialized value using the specified name. It is equivalent to <code>create(name, NaN)</code> .
<code>Value create(name, value)</code>	Creates a value using the specified name. It is equivalent to <code>create(name, value, null)</code> .
<code>Value create(name, value, unit)</code>	Creates a value using the specified name and unit. If script reference defines output mapping, the name will be transformed to the reference output name.
<code>void discard(value)</code>	Discards a previously created value.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <script id="capacity-decode" in="Capacity" out="Capacity">
    <![CDATA[
      v = ctx.str(Capacity);
      if(v.matches("[0-9\\.]+)(\\_GB)?")) {
        ctx.create("Capacity",
          v.replaceAll("[0-9\\.]+)(\\_GB)?", "$1")*1024*1024*1024,
          "Bytes");
      } else if(v.matches("[0-9\\.]+)(\\_MB)?")) {
        ctx.create("Capacity",
          v.replaceAll("[0-9\\.]+)(\\_MB)?", "$1")*1024*1024,
          "Bytes");
      } else if(v.matches("[0-9\\.]+)(\\_KB)?")) {
        ctx.create("Capacity",
          v.replaceAll("[0-9\\.]+)(\\_KB)?", "$1")*1024,
          "Bytes");
      } else if(v.matches("[0-9\\.]+)_*")) {
        ctx.create("Capacity",
          v.replaceAll("[0-9\\.]+)_*", "$1")*1,
          "Bytes");
      }
    ]]>
  </script>
  <!-- ... -->
</domain>

```

This sample script convert a capacity string into a numeric value. The capacity string is expected to be a number followed by a unit. Please note the implicit string to number conversion in `v.replaceAll("[0-9\\.]+) *", "$1")*1`.

The script may be applied with the following reference:

```
<scripted-value script-id="capacity-decode" in="TotalSize" out="TotalSize" />
```

The Capacity parameter will be dynamically mapped to TotalSize attribute. The created value will be automatically renamed from "Capacity" to TotalSize. Note that the mappings doesn't prevent script from creating more value, however, they will not be automatically renamed.

Another usefull example is the `delta` script. It shows how you can store data across polling cycles to compute deltas.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <script id="rate" in="value_unit" out="rate">
    <![CDATA[
      previousValue = ctx.ifetch(rate);
      previousTimestamp = ctx.itimestored(rate);

      currentValue = ctx.num(value);
      currentTimestamp = ctx.time();

      if (previousValue != null) {
        newRate = (currentValue-previousValue) / (currentTimestamp-previousTimestamp);
        if (newRate >= 0) {
          ctx.create(rate, newRate, unit);
        }
      }
      ctx.istore(rate, currentValue);
    ]]>
  </script>
  <!-- ... -->
</domain>
```

5.6 Scripted properties

Scripted properties are scripts which generate properties. Properties are generated with the `create` methods. Every created property will be automatically sent along with values without any other operation unless discarded with the `discard` method. These scripts can access the following methods from `ctx`:

Value <code>create(name, value)</code>	Creates a property using the specified name. It is equivalent to <code>create(name, value, null)</code> .
--	---

Value <code>create(name, value, appendWith)</code>	Creates a property using the specified name and append it using the specified separator. If script reference defines output mapping, the name will be transformed to the reference output name.
void <code>discard(value)</code>	Discards a previously created property.

A scripted-property element can have a `make-variable` attribute. It has exactly the same meaning as in property elements (see [Fake variable generation](#)).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <script id="non-empty" in="a1_a2" out="prop">
    <![CDATA[
      if(ctx.str(a1).length() > 0) {
        ctx.create("prop", a1);
      } else if(ctx.str(a2).length() > 0) {
        ctx.create("prop", a2);
      }
    ]]>
  </script>
  <!-- ... -->
</domain>
```

The script may be applied with the following reference:

```
<scripted-property script-id="non-empty" in="DeviceID_Description" out="part" />
```

This will create the `part` property from the first non empty value of `DeviceID` or `Description`. If both are empty, the property will not be created.

5.7 Instance filter scripts

Filter scripts has no special functionality but must return `true` or `false`. The returned value is the value of the last script statement (there is no `return` keyword).

The main difference between a pre-filter-script and a post-filter-script is that a pre-filter script can't have any input because it is executed *before* instance attributes polling. The only contextual information it can access is the *EMC² Smarts InCharge* instance name (smInstance variable).

A post-filter script can override APG instance name. This is simply done overwriting the apgInstance variable (which normally contains the previously set APG instance name).

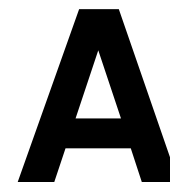
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE domain SYSTEM "indicators.dtd">
<domain>
  <!-- ... -->
  <script id="physical-device">
    <![CDATA[
      smInstance.indexOf("physical") >= 0
    ]]>
  </script>

  <script id="fast-eth-iface" in="Type_MaxSpeed">
    <![CDATA[
      apgInstance = "FAST-ETH-" + apgInstance;
      ctx.num(MaxSpeed) >= 100000000
      && ctx.str(Type).equals("ETHERNETCSMACD")
    ]]>
  </script>

  <!-- ... -->

  <data class="Interface">
    <pre-filter-script script-id="physical-device" />
    <post-filter-script script-id="fast-eth-iface" in="Type_MaxSpeed" />
    <property name="Name" rename="part"/>
    <value name="Status" rename="Availability" unit="%">
      <transformation match="UP" replace="100.0" />
      <transformation match="DOWN" replace="0.0" />
    </value>
  </data>
  <!-- ... -->
</domain>
```

This is a pretty complete example where we first filter interfaces which instance name contains the string physical and then which Type is ETHERNETCSMACD and MaxSpeed $\geq$ 100 Mbits.



Configuration hints

1 *EMC² Smarts InCharge* settings

To enhance *EMC² Smarts InCharge* data collecting, you should take care of some specific domain settings.

1.1 Polling period

Check your polling settings to be consistent with collector settings. Indeed, the collector will retrieve data from *EMC² Smarts InCharge* every 4 minutes (this is the default settings, you can change it afterward) but if *EMC² Smarts InCharge* polling is set to an higher value, you will have some duplicated entries in the database. To change this setting, go to the *Polling and Threshold* panel and set every polling period according to your desired accuracy.

1.2 Interface duplex mode setting

Check your *EMC² Smarts InCharge* configuration to enable the interface/port duplex mode retrieval. Indeed, *EMC² Smarts InCharge* needs it to compute the current interface/port utilization which is collected by *APG*. To do so, edit the `discovery.conf` file from *EMC² Smarts InCharge* configuration (discovery directory) and set `DuplexAssumed` to *true*.