CA Unified Infrastructure Management

Simple Local Probe Example

V1.0

ca technologies

Legal Notices

This online help system (the "System") is for your informational purposes only and is subject to change or withdrawal by CA at any time.

This System may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA. This System is confidential and proprietary information of CA and protected by the copyright laws of the United States and international treaties. This System may not be disclosed by you or used for any purpose other than as may be permitted in a separate agreement between you and CA governing your use of the CA software to which the System relates (the “CA Software”). Such agreement is not modified in any way by the terms of this notice.

Notwithstanding the foregoing, if you are a licensed user of the CA Software you may make one copy of the System for internal use by you and your employees, provided that all CA copyright notices and legends are affixed to the reproduced copy.

The right to make a copy of the System is limited to the period during which the license for the CA Software remains in full force and effect. Should the license terminate for any reason, it shall be your responsibility to certify in writing to CA that all copies and partial copies of the System have been destroyed.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS SYSTEM “AS IS” WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS SYSTEM, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The manufacturer of this System is CA.

Provided with “Restricted Rights.” Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

Copyright © 2015 CA. All rights reserved. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Legal information on third-party and public domain software used in the Nimsoft Monitor solution is documented in Nimsoft Monitor Third-Party Licenses and Terms of Use *(*http://docs.nimsoft.com/prodhelp/en_US/Library/Legal.html*).*

Contact CA

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback About Product Documentation

Send comments or questions about CA Technologies Nimsoft product documentation to nimsoft.techpubs@ca.com.

To provide feedback about general CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

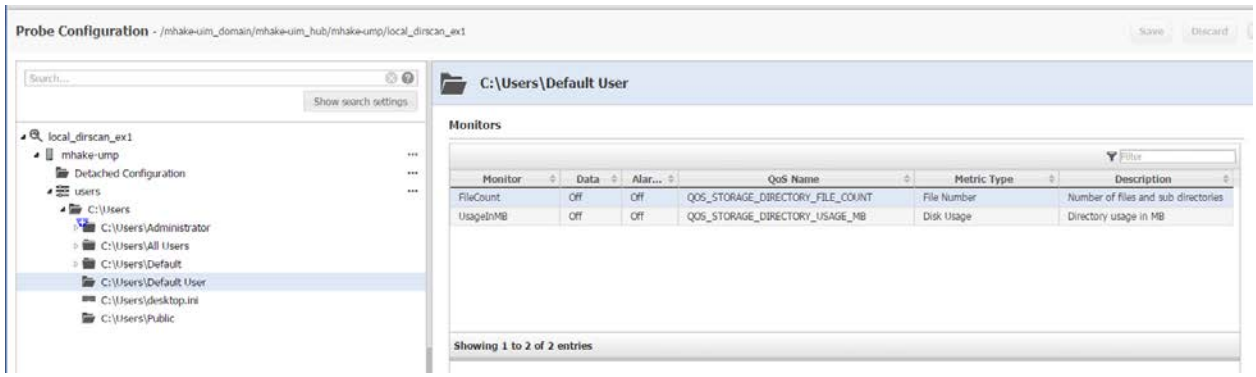
Create a Simple Local Probe

This article provides an example scenario for creating a simple local probe. The Local Directory Scan probe is a simple local probe that takes three basic measurements against a specified folder on the local file system. The final source code for this example (local_dirscan_ex1) is in the Probe-SDK. For information about the Probe-SDK, see the Probe Framework SDK Guide.

Final Probe Implementation

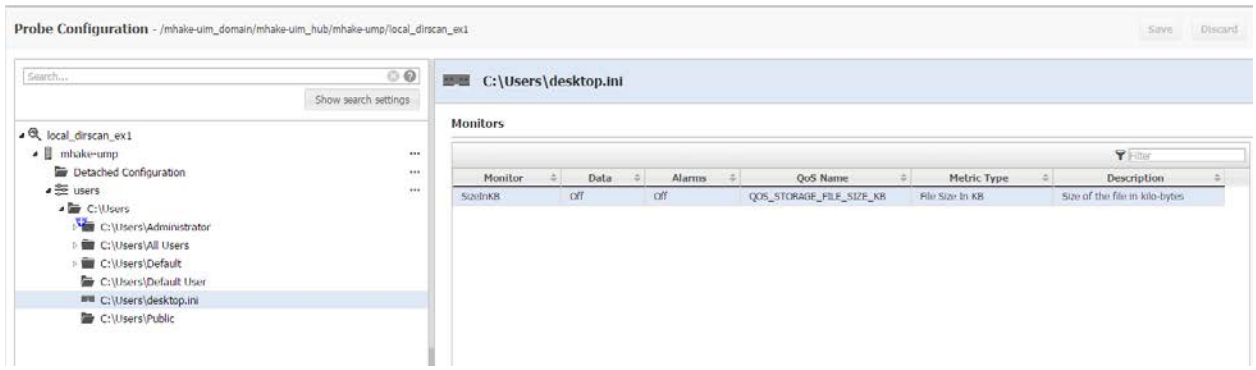
This section describes the final probe that you will create with this example. This probe can be deployed and configured to scan a local directory and produce an inventory. The full example of the Local Directory Scan (local_dirscan_ex1) probe is included in the Probe-SDK.

The following image shows the inventory pane and the details pane for the probe.

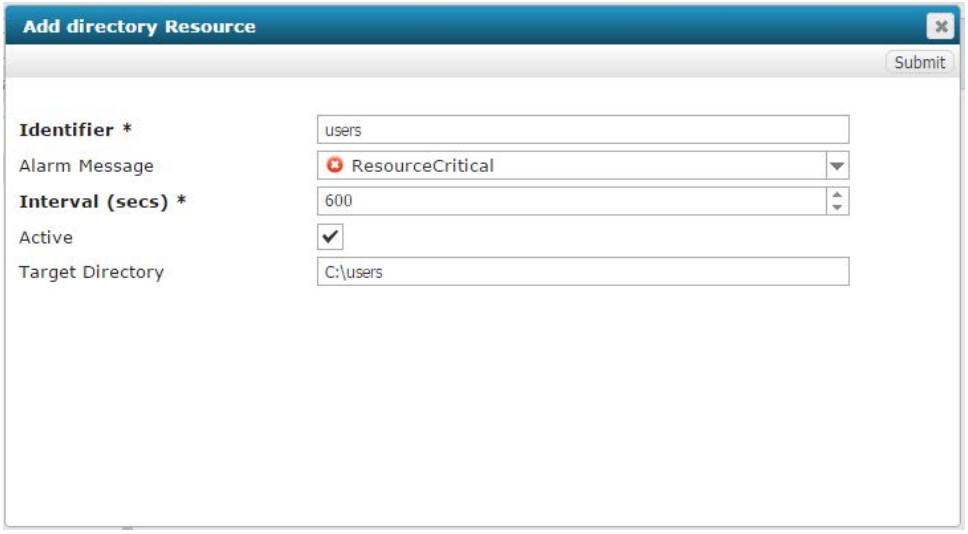


The inventory pane on the left shows the structure of the "C:\Users" directory on the local system. In the right we can see that we can apply measurements for the number of files and the total directory size.

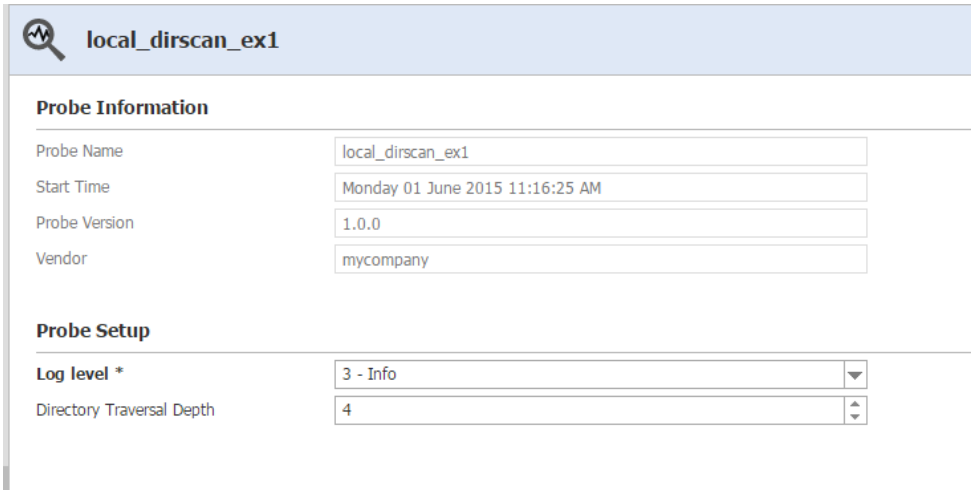
The following image shows the measurement available for a file. The only measurement is the file size in KB.



The following image shows the resource configuration screen. This screen allows the end user to configure a directory for monitoring.



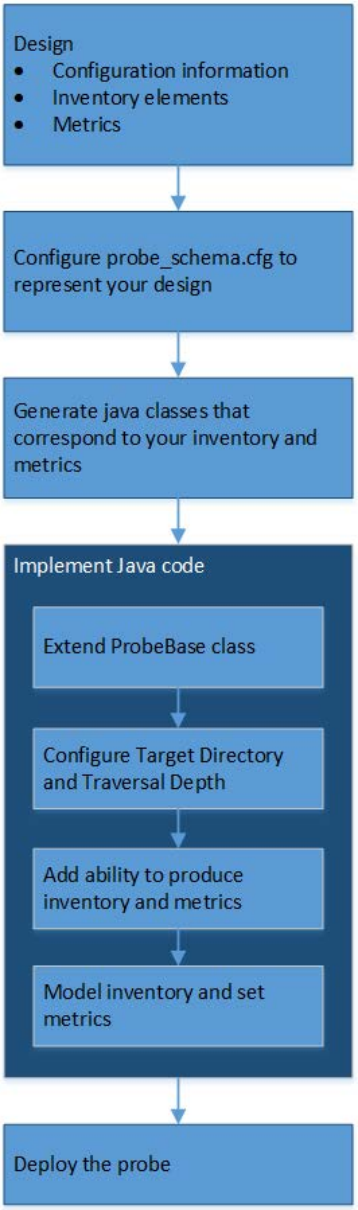
The following image shows how the end user sets the global configuration settings.



Note the Directory Traversal Depth property in the Probe Setup section. This property limits how deep the probe monitors the directory structure for all configured resources.

Development Process

The following diagram shows the process for developing the probe.



Design Requirements

In this example, the Local Directory Scan probe must fulfill the following requirements:

- Monitor the size of a specified file system directory in megabytes.
- Monitor the number of files and subdirectories in a given directory.
- Monitor the size of a specified file in kilobytes.

Probe Design

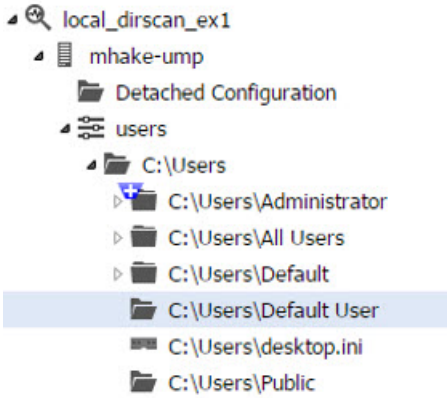
Before you create any code or modify any files, you must define what inventory the probe monitors and what metrics the probe collects. The probe needs to collect information on directories and files, so you need to model these elements. You need to model:

- Inventory elements for directories and files. You need these elements since the probe must monitor the files and directories in a file system.
- Two metrics for the directory inventory element. You need a metric to measure directory size and a metric to measure file count.
- A metric for the file inventory element. You need this metric to measure file size.

The end user must have the ability to identify what directory to monitor and manage the depth of monitoring within the directory structure. To provide these settings, you need the following configuration properties:

- Target Directory: You need this configuration property so a user can identify the target directory to monitor on the resource. This property is configured for each monitoring target.
- Traversal Depth: You need this configuration property to limit the depth of monitoring within a directory structure. This property is configured as part of the probe setup configuration so the probe applies this setting globally to all monitoring targets.

Both of these properties are probe specific and are not required by the Probe-SDK.



Define Inventory Elements

The Probe Framework provides an assortment of Java classes that you use to represent your inventory elements. Refer to the JavaDoc included with the Probe Framework and refer to the com.nimsoft.probe.framework.devkit.inventory package.

The following image shows the JavaDoc page for com.nimsoft.probe.framework.devkit.inventory.

Package com.nimsoft.probe.framework.devkit.inventory

This package contains all the classes the Probe Framework provides for inventory modeling and Metric reporting.

See: Description

Class Summary

Class	Description
ClusterElement	Inventory modeling class that represents a group of compute resources in a virtual environment.
CollectionElement	Inventory modeling class for other metric groupings and aggregates.
ComputerSystemElement	Inventory modeling class that represents a `ComputerSystem Element`
DiskPartitionElement	Inventory modeling class that represents a `DiskPartition Element`
Element	Element(s) are used to model a probe's inventory of components with metrics ...
Element.ChildParentRelationship	
Element.ElementCounter	
Element.PropertyDefinitions	
Element.TypeDefinition	
FileElement	Inventory modeling class that represents a file or directory
Folder	Folders are a pure UI construct and are only for organizational purposes.
FolderView	Deprecated
HardwareElement	Inventory modeling class that represents a `RunningHardware Element`
HypervisorManagerElement	Inventory modeling class that represents a `HypervisorManager Element`
IPDeviceElement	Inventory modeling class that represents a `IPDevice Element`
MemoryElement	Inventory modeling class that represents a `Memory Element`
NetworkInterfaceElement	Inventory modeling class that represents a `NetworkInterface Element`
PortElement	Inventory modeling class that represents a `Port Element`
ProcessorElement	Inventory modeling class that represents a `Processor Element`
RouterElement	Inventory modeling class that represents a `Router Element`
SoftwareElement	Inventory modeling class that represents a `RunningSoftware Element`
StorageVolumeElement	Inventory modeling class that represents a `StorageVolume Element`
SwitchElement	Inventory modeling class that represents a `Switch Element`
VirtualSystemElement	Inventory modeling class that represents a `VirtualSystem Element`

Package com.nimsoft.probe.framework.devkit.inventory Description

This package contains all the classes the Probe Framework provides for inventory modeling and Metric reporting. All of the classes in this package that end in 'Element' are valid to use for the element-types you declare in probe_schema.cfg.

When you model your inventory element after one of the Element's contained here (in probe_schema.cfg) you will also attach the Metric Definitions you wish to report.

Note: This package also contains `Folder` which is a pure UI construct and are only for organizational purposes. Folders should not include metrics.

The classes in this package represent many common concepts in computing and networking. Select the class or classes that is the best match for your inventory model. The probe developer must select classes from this package that best represent the inventory elements they wish to model.

You must model files and directories in this example. Look at the Javadoc for available modeling classes, you can see that the 'FileElement' class is for representing both files and directories. Use 'FileElement' to model the StorageFile and StorageDirectory elements.

Define Metrics for Inventory Elements

Both the StorageDirectory and StorageFile elements require metrics. Based on the probe requirements, the StorageDirectory element needs metrics to measure directory size and file count. The StorageFile element needs a metric to measure file size.

Review the MetricTypesWithUnits file in the docs directory. This file contains a list of available metric types. In this example, select the metrics for Directory usage in MB, Number of files and sub directories, and Size of the file in kilo-bytes.

Note: See the Terminology and QoS Metric Types in the Probe Framework SDK Guide for more information about metrics.

You define the following five parameters for each metric:

- QoS Name
- Metric Type
- Units
- Description
- Label
- Active Flag

Here is an example of the information for the StorageDirectory metrics:

- Directory usage in MB
  QOS Name: QOS_STORAGE_DIRECTORY_USAGE_MB
  Description: Directory usage in MB
  Metric Type: 1.1:2
  Unit: MB
  Label: UsageInMB
  Active: yes
- Number of files and sub directories
  QOS Name: QOS_STORAGE_DIRECTORY_FILE_COUNT
  Description: Number of files and sub directories
  Metric Type: 1.10:1
  Unit: count
  Label: FileCount
  Active: yes

Here is an example of the information for the StorageFile metric:

- Size of the file in kilo-bytes
  QOS Name: QOS_STORAGE_FILE_SIZE_KB
  Description: Size of the file in kilo-bytes
  Metric Type: 1.10:19
  Unit: KB

Label: SizeInKB
Active: yes

Example probe_schema.cfg File

Configure the **probe_schema.cfg** file when the design work for the inventory and metrics is complete. You define the element_types (Inventory) and the associated QoS metric types in this file.

Notes:

- The RESOURCE section and the ResourceResponseTime metric is a default metric that must be included with all probes. This section defines a monitorable target.
- The base_element_type declaration in the inventory types StorageDirectory and StorageFile translate to auto generated java classes that extend FileElement.
- An icon for memory is specified for StorageFile. This is the icon that will appear for that element in the probe configuration GUI

The following example shows the file:

probe_schema.cfg

```
<probe_schema>
    <element_types>
        <RESOURCE>
            <properties>
            </properties>
            <qos_metric_types>
                <ResourceResponseTime>
                    active = yes
                    unit = MilliSeconds
                    descr = Duration time in milli-seconds of last resource collection cycle.
                    name_label = ResourceResponseTime
                    metric_type = 1.10:14
                    qos_name = QOS_RESOURCE_RESPONSE_TIME
                </ResourceResponseTime>
            </qos_metric_types>
        </RESOURCE>
        <StorageDirectory>
            <properties>
                base_element_type = FileElement
            </properties>
            <qos_metric_types>
                <UsageInMB>
                    unit = MB
                    descr = Directory usage in MB
                    active = yes
                    name_label = UsageInMB
                    metric_type = 1.1:2
                    qos_name = QOS_STORAGE_DIRECTORY_USAGE_MB
                </UsageInMB>
                <FileCount>
                    unit = count
                    descr = Number of files and sub directories
                    active = yes
                    name_label = FileCount
                    metric_type = 1.10:1
                    qos_name = QOS_STORAGE_DIRECTORY_FILE_COUNT
                </FileCount>
            </qos_metric_types>
        </StorageDirectory>
        <StorageFile>
```

```
            <properties>
                base_element_type = FileElement
                icon = memory
            </properties>
            <qos_metric_types>
                <SizeInKB>
                    unit = KB
                    descr = Size of the file in kilo-bytes
                    active = yes
                    name_label = SizeInKB
                    metric_type = 1.10:19
                    qos_name = QOS_STORAGE_FILE_SIZE_KB
                </SizeInKB>
            </qos_metric_types>
        </StorageFile>
    </element_types>
</probe_schema>
```

Auto Generate Java Classes

The **probe_schema.cfg** files defines the inventory and metrics. To create the connection between the Java code and the inventory, you must create equivalent Java classes. The Probe-SDK provides template examples that you can use to create a probe project. When you create a project from one of the templates, the project is preconfigured to use Maven as the build tool.

Note: Classes are auto-generated. Do not make any modifications to the classes.

Run the **mvn install** command to start a standard Maven build and invoke the plugin to generate the Java classes based on the information in the **probe_schema.cfg** file. In this example, the command auto-generated the following java classes:

StorageDirectory.java

```
/*******************  G E N E R A T E D  C O D E  ********  A V O I D  C H A N G I N
G   **********************/
/*** This is generated source code which will change with PF-SDK evolution.  Avoid changing for
easy maintenance. ***/
public class StorageDirectory extends FileElement {
    public static StorageDirectory addInstance(IInventoryDataset ds, EntityId entityId, String
nameLabel, Element... guiParents) throws NimException, InterruptedException {
        StorageDirectory ele = new StorageDirectory(entityId, nameLabel, guiParents);
        ds.addItem(ele);
        return ele;
    }
    protected StorageDirectory(EntityId id, String labelName, Element... guiParents) {
        super(ElementDef.Factory.defineElementType("StorageDirectory"), id, labelName,
guiParents);

    }

    public static final MetricDef FileCount = new MetricDef("FileCount", "Number of files and sub
directories", "1.10:1", "QOS_STORAGE_DIRECTORY_FILE_COUNT"); // FileCount

    public static final MetricDef UsageInMB = new MetricDef("UsageInMB", "Directory usage in MB",
"1.1:2", "QOS_STORAGE_DIRECTORY_USAGE_MB"); // UsageInMB

}
```

StorageFile.java

```
/********************  G E N E R A T E D  C O D E  ********  A V O I D  C H A N G I N
G   ************************/
/*** This is generated source code which will change with PF-SDK evolution.  Avoid changing for
easy maintenance. ***/
public class StorageFile extends FileElement {
    public static StorageFile addInstance(IInventoryDataset ds, EntityId entityId, String
nameLabel, Element... guiParents) throws NimException, InterruptedException {
        StorageFile ele = new StorageFile(entityId, nameLabel, guiParents);
        ds.addItem(ele);
        return ele;
    }
    protected StorageFile(EntityId id, String labelName, Element... guiParents) {
        super(ElementDef.Factory.defineElementType("StorageFile"), id, labelName, guiParents);
    }
    public static final MetricDef SizeInKB = new MetricDef("SizeInKB", "Size of the file in kilo-
bytes", "1.10:19", "QOS_STORAGE_FILE_SIZE_KB"); // SizeInKB
}
```

Implement Probe Java Code

To use the Probe-SDK to build a viable probe you will always extend the class **ProbeBase.java**. If your probe is going to produce inventory and metrics you will also need to implement the **IProbeInventoryCollection** interface.

Class declaration

```
public class local_dirscan_ex1Probe extends ProbeBase implements IProbeInventoryCollection {

}
```

Main Method and Constructor

Add a **main()** method and a probe constructor to allow the probe framework to start up the probe and register the probe with the UIM bus. Every probe is a standalone Java program that must start itself up and register itself with the bus. The Probe-SDK provides all the logic for doing this in the {@code ProbeBase} base class. So all you must do when implementing a probe is create a simple {@code main()} method that invokes the proper life cycle methods in {@code ProbeBase}.

Main method - Every probe is a standalone Java program that must start itself up and register itself with the bus.

Constructor - You must implement a constructor that calls the super constructor and passes in the parameters shown in the following example. Call your constructor from the main() method. You can modify the method signature to suit your needs. For example if you wanted to pass some additional arguments.

main

```
    public class local_dirscan_ex1Probe extends ProbeBase implements IProbeInventoryCollection {
    /*
     * Probe Name, Version, and Vendor are required when initializing the probe.
     */
    public final static  String PROBE_NAME = "local_dirscan_ex1";
    public static final  String PROBE_VERSION = MvnPomVersion.get("com.nimsoft", PROBE_NAME);
    public static final  String PROBE_VENDOR = "mycompany";
```

```
    /**
     * Every probe is a stand alone Java program that must start itself up and
     * register itself with the bus. The Probe Framework provides all the logic
     * for doing this in the {@code ProbeBase} base class. So all we must do when
     * implementing a probe is create a simple {@code main()} method that invokes
     * the proper life cycle methods in {@code ProbeBase}
     *
     * @param args
     */
    public static void main(final String[] args) {
        try {
            ProbeBase.initLogging(args);
            local_dirscan_ex1Probe probeProcess = new local_dirscan_ex1Probe(args);
            Log.info("Probe " + PROBE_NAME + " startup");
            probeProcess.execute();
        } catch (final Exception e) {
            ProbeBase.reportProbeStartupError(e, PROBE_NAME);
        }
    }

    /**
     * You must implement a constructor that calls the super constructor
     * and passes in the parameters shown below. You will call your
     * constructor from the main() method and you may modify the method
     * signature to suit your needs. For example if you wanted to pass some
     * additional arguments.
     * @throws NimException
     */
    public local_dirscan_ex1Probe(String[] args) throws NimException {
        super(args, PROBE_NAME, PROBE_VERSION, PROBE_VENDOR);
        // Indicate that this is a local probe so that it is displayed correctly in the UI
        setIsLocalProbeFlag(true);
        // Helper method to provide simple pathname values in UI instead of relative path values
        GenNodeFactory.useSimpleBaseNamesInsteadOfRelativePaths = true;
    }

}
```

Configuration with addDefaultProbeconfigruationToGraph()

Enable the configuration properties (Target Directory and Traversal Depth) to appear in the probe configuration GUI.

This is where you configure what you want to display in the probe configuration UI. There are 2 different areas of configuration, the "Setup" and the "Resource" properties. In the following example the "Directory Traversal Depth" to the probe "Setup" options are added. Also add the "Target Directory" to the "Resource" properties.

Note: Implementing this method is optional. If your probe does not require the ability to specify configuration options in the Probe Configuration UI then you may skip implementing this method.

You must override the ProbeBase method **addDefaultProbeConfigurationToGraph()** as follows:

addDefaultProbeConfigurationToGraph

```
/**
 * This is where you configure what you want to display in the probe configuration UI.
 * There are 2 different areas of configuration, the "Setup" and the "Resource" properties.
 * In this example we add the "Directory Traversal Depth" to the probe "Setup" options. We
 * also add the "Target Directory" to the "Resource" properties.
 *
 * Note: Implementing this method is optional. If your probe does not require the ability to
 * specify configuration options in the Probe Configuration UI then you may skip implementing
 * this method.
 */
```

```
@Override
public void addDefaultProbeConfigurationToGraph() {
    // To enable the user to add resources for the probe in the probe oriented configuration UI,
    // the following adds a resource definition to the profile
    ElementDef resDef = ElementDef.getElementDef("RESOURCE");
    resDef.addStandardAction(IProbeResourceTypeInfo.StandardActionType.DeleteProfileAction);
    resDef.addStandardAction(IProbeResourceTypeInfo.StandardActionType.VerifySelectionAction,
"Verify Directory Configuration");
    resDef.addStandardAction(IProbeResourceTypeInfo.StandardActionType.AddProfileActionOnProbe,
"Add directory Resource");
    resDef.setIsAResourceInfo(true);
    // add SETUP properties
    CtdPropertyDefinitionsList setupPropDefs =
CtdPropertyDefinitionsList.createCtdPropertyDefinitionsList("SETUP", getGraph());
    // adds the traversal depth property, with a default value
    setupPropDefs.addIntegerPropertyUsingEditField(TRAVERSAL_DEPTH_PROP, "Directory Traversal
Depth", TRAVERSAL_DEPTH_DEFAULT);

    // Next set the properties that will be available when a new profile is created in the probe
configuration UI
    CtdPropertyDefinitionsList profilePropDefs =
CtdPropertyDefinitionsList.createCtdPropertyDefinitionsList("RESOURCE", getGraph());
    // Add an identifier (essentially the profile's name)
    profilePropDefs.addStandardIdentifierProperty();
    // Currently required, select the alarm message to alarm with when the resource is
unavailable
    profilePropDefs.addStandardAlarmMessageProperty();
    // Add the polling interval property
    profilePropDefs.addStandardIntervalProperty();
    // Add a checkbox to select whether the profile is active or inactive (inactive ones are not
used)
    profilePropDefs.addStandardActiveProperty();
    // Add a property for selecting the directory to be monitored
    profilePropDefs.addStringPropertyUsingEditField(TARGET_DIR_PROP, "Target Directory",
"C:\\tmp");
    profilePropDefs.setCfgPathname(TARGET_DIR_PROP, "properties/"+TARGET_DIR_PROP);
    // Finally, make the call to update the default configuration and get these settings
published to UIM so that they
    // appear correctly in the probe configuration UI
    super.addDefaultProbeConfigurationToGraph();
}
```

Implement IProbeInventoryCollection

The methods necessary to produce the inventory and metrics are specified in the IProbeInventoryCollection interface. The methods specified in this interface are called **testResource()** and **getUpdatedInventory().**

```
* The method should be implemented to validate the probe configuration. For example
 * if the configuration specifies remote system connectivity information. If unable
 * to successfully verify the configuration then this method should throw an {@code Exception}
 * with a message about the nature of the problem.
 * </p>
 * Note: The tests performed here do not need to be limited to connectivity. You should verify
 * anything and everything related to your probe configuration.
```

The implementation of the methods for this probe are as follows:

IProbeInventoryCollection methods

```
/**
 * Allows the user to test a profile configuration from the UI by using
 * the pull down: Actions->verify.
 * </p>
 * All probe framework probes must implement this method.
 * </p>
```

```
 * The method should be implemented to validate the probe configuration. For example
 * if the configuration specifies remote system connectivity information. If unable
 * to successfully verify the configuration then this method should throw an {@code Exception}
 * with a message about the nature of the problem.
 * </p>
 * Note: The tests performed here do not need to be limited to connectivity. You should verify
 * anything and everything related to your probe configuration.
 *
 * Note: The methods {@code testResource()} and {@code getUpdatedInventory()} are both from the
 * interface {@code IProbeInventoryCollection}. You will need to implement these methods if your
 * probe implements {@code IProbeInventoryCollection}. Please see the JavaDoc on that interface
 * for more details.
 *
 * @param resource Configuration information.
 *
 * @return IInventoryDataset Optional. This is reserved for a future enhancement for an advanced
probe
 * to provide additional resource configuration information. However this is presently not used,
 * so the best practice is to return {@code null}
 *
 * @throws Exception if any errors are encountered during the testing of the configuration.
 */
@Override
public IInventoryDataset testResource(ResourceConfig res) throws NimException,
InterruptedException {
    Log.info("==== testResource: " + res.getName());

    // Test that target directory configuration is valid
    // If it not we throw an exception stating why
    String targetDir = res.getResourceProperty(TARGET_DIR_PROP);
    if (targetDir == null || targetDir.isEmpty()) {
        String errMsg = "No Target Directory Specified.";
        Log.error("testResource: " + res.getName() + "   " + errMsg);
        throw new NimException(NimException.E_ERROR, errMsg);
    }
    File root = new File(targetDir);
    if (!root.exists()) {
        String errMsg = "Target Directory Does Not Exist: " + targetDir;
        Log.error("testResource: " + res.getName() + "   " + errMsg);
        throw new NimException(NimException.E_ERROR, errMsg);
    }

    // If we get to here then our tests were successful. Since we dont have
    // any advanced information we wish to return we can simply return null
    return null;
}

/**
 * This is called by the framework on the inventory collection cycle. In this method
 * we construct the inventory, and attach metrics. We always attach all metrics, and the
 * framework will determine which ones to publish based on how the probe is configured.
 *
 * We return data to the framework in both the returned InventoryDataset, AND the passed
 * in ResourceConfig. Every inventory Element you create will be attached to the
InventoryDataset,
 * those Elements must also be constructed in a hierarchy attached to ResourceConfig.
 *
 * Note: The methods {@code testResource()} and {@code getUpdatedInventory()} are both from the
 * interface {@code IProbeInventoryCollection}. You will need to implement these methods if your
 * probe implements {@code IProbeInventoryCollection}. Please see the JavaDoc on that interface
 * for more details.
 */
@Override
public IInventoryDataset getUpdatedInventory(ResourceConfig resourceConfig, IInventoryDataset
previousDataset) throws NimException, InterruptedException {
    // A recommended best practice is to read configuration information
    // on each call to getUpdatedInventory(). This ensures configuration changes
    // take effect without the need for a full restart of the probe.
    // Also, please note that the configuration information is cached by the
    // framework, so there is very low overhead here.
    int counter = resourceConfig.updateCounter;
```

```
    String targetDir = resourceConfig.getResourceProperty(TARGET_DIR_PROP);
    int traversalDepth = NimConfig.getInstance().getValueAsInt("/setup", TRAVERSAL_DEPTH_PROP,
TRAVERSAL_DEPTH_DEFAULT);

    // Build the model using the ResourceConfig as the root element
    InventoryDataset inventoryDataset = new InventoryDataset(resourceConfig);
    Log.info("==== Begin getUpdatedInventory: Pass-" + counter + "   " +
resourceConfig.getName());
    buildInventory(new File(targetDir), traversalDepth, inventoryDataset, resourceConfig);
    return inventoryDataset;
}
```

Model Inventory and Set Metrics

If you review the getUpdatedInventory() method above, you see that the method is building up and returning an InventoryDataset. Be aware that the details are hidden in a method called recursivelyBuildInventory(). The following list describes what information you must return from the getUpdatedInventory() method.

For each inventory element (StorageDirectories and StorageFiles):

- Use the provided static factory method to create a new instance of the element, add it to the InventoryDataset, and also create a relationship with its parent. For example:

  StorageDirectory storageDirectory = StorageDirectory.addInstance(inventoryDataset, new EntityId(absolutePath), absolutePath, parent);

  Note: For parent in the above line, use the parent StorageDirectory, However, the parent that is passed into the method for the top level directory in the hierarchy is ResourceConfig. When modeling your inventory hierarchy the top level parent object is the ResourceConfig.

- Set all of the metrics you defined for that element using the setMetric() method on your inventory element instance. For example:

  storageDirectory.setMetric(StorageDirectory.UsageInMB, FileUtils.sizeOfDirectory(fileOrDirectory) / BytesPerMB);

Here is the pseudo code for constructing inventory in the getUpdatedInventory() method

- Construct a new InventoryDataset
- For each item in your inventory:
  - Create a new instance of the inventory element using the static factory 'addInstance()' method. You pass a reference of the InventoryDataset as well as the parent object. For the top most item or items in your inventory structure the parent will be the ResourceConfig object.
  - For each metric on this inventory element
    - Call the 'setMetric()' method on the element, passing the metric name and the value.

Return the newly created InventoryDataset that now contains a collection of all inventory items and metrics, as well as the ResourceConfig which contains the parent/child structure information.

Now that you know what information must be returned, review the following examples of the BuildInventory() method.

buildInventory

```
   /**
 * Traverse the directory structure and build our inventory tree and
 * associated metrics.
 * @param fileOrDirectory A file or directory to add to inventory and gather metrics on
 * @param depth The present recursion depth. This limits how deep the recursion will go.
 * @param inventoryDataset This is both an input an output. As new Elements are discovered, they
 * will be added to the InventoryDataset
 * @param parent The parent inventory item. The root parent will be ResourceConfig.
 * @param isRoot indicates if this is the root directory in the tree. Necessary to know so we can
construct the lable correctly
 * @throws NimException
 * @throws InterruptedException if a probe shutdown request from the framework is detected
 */
private void buildInventory(File fileOrDirectory, int depth, IInventoryDataset inventoryDataset,
Element parent) throws NimException, InterruptedException {
    // If your probe has a long running collection cycle it is a recommended
    // best practice to periodically call handleInterrupt() to give the probe an
    // opportunity to check if it has received a shutdown request. This particular
    // example is not long running, but we still demonstrate the practice here
    handleInterrupt();
    if (fileOrDirectory.isFile()) {
        addFileAndMetricsToInventory(fileOrDirectory, inventoryDataset, parent);
    } else if (fileOrDirectory.isDirectory()) {
        addDirectoryAndMetricsToInventory(fileOrDirectory, depth, inventoryDataset, parent);
    }
}

/**
 * Model the passed in File as our StorageFile data type. Connect it to the passed in parent,
 * add it to the InventoryDataset and add metrics.
 * @param file
 * @param inventoryDataset
 * @param parent
 * @throws NimException
 * @throws InterruptedException
 */
private void addFileAndMetricsToInventory(File file,IInventoryDataset inventoryDataset, Element
parent) throws NimException, InterruptedException{
    String absolutePath = file.getAbsolutePath();
    // Use static factory method to create a new instance of our model object and add it to
parent and InventoryDataset.
    StorageFile storageFile = StorageFile.addInstance(inventoryDataset, new
EntityId(absolutePath), absolutePath, parent);
    // Set the metrics this probe collects on the StorageFile. We always set them all,
    // and let the framework determine which ones to publish to the bus.
    storageFile.setMetric(StorageFile.SizeInKB, new Double((double) (file.length()) / 1024));
}

/**
 * Model the passed in directory as our StorageDirectory data type. Connect it to the passed in
 * parent and add it to the InventoryDataset. Also add metrics.
 * @param directory
 * @param depth
 * @param inventoryDataset
 * @param parent
 * @throws NimException
 * @throws InterruptedException
 */
private void addDirectoryAndMetricsToInventory(File directory, int depth, IInventoryDataset
inventoryDataset, Element parent) throws NimException, InterruptedException{
    String absolutePath = directory.getAbsolutePath();
```

```
    // Use static factory method to create a new instance of our model object and add it to
parent and InventoryDataset.
    StorageDirectory storageDirectory = StorageDirectory.addInstance(inventoryDataset, new
EntityId(absolutePath), absolutePath, parent);
    // Set the metrics this probe collects on the StorageDirectory. We always set them all,
    // and let the framework determine which ones to publish to the bus.
    storageDirectory.setMetric(StorageDirectory.UsageInMB, FileUtils.sizeOfDirectory(directory) /
1024000);
    storageDirectory.setMetric(StorageDirectory.FileCount, getFileCount(directory));
    if (depth > 0) {
        // We have not yet hit the max traversal depth, so process the contents of this directory
        File[] files = directory.listFiles();
        if (files != null) {
            for(File f : files){
                // Recursively call buildInventory to process this sub directory. Note that
                // we decrement the depth counter to limit how deep we will recurse.
                // Also note how we pass the newly created storageDirectory as the parent. This
                // gives us the tree structure we desire. All items found will be children of
this
                // directory
                buildInventory(f, depth--, inventoryDataset, storageDirectory);
            }
        }
    }
}
```

Javadocs are included in the pfdevkit-dist.zip package. Use the Javadocs to view the methods for ProbeBase and InventoryDataset.