Foglight® 7.3.0

Creating Actions Field Guide

If you have any questions regarding your potential use of this material, contact:

Quest Software Inc.
Attn: LEGAL Dept.
4 Polaris Way
Aliso Viejo, CA 92656

Refer to our website (https://www.quest.com) for regional and international office information.

Patents

Quest Software is proud of our advanced technology. Patents and pending patents may apply to this product. For the most current information about applicable patents for this product, please visit our website at https://www.quest.com/legal.

Trademarks

their respective owners.

Legend

WARNING: A WARNING icon indicates a potential for property damage, personal injury, or death.

CAUTION: A CAUTION icon indicates potential damage to hardware or loss of data if instructions are not followed.

IMPORTANT NOTE, **NOTE**, **TIP**, **MOBILE**, or **VIDEO:** An information icon indicates supporting information.

Foglight Creating Actions Field Guide
Foglight Version- 7.3.0

1

Adding Custom Action Types to Foglight

This *Creating Actions Field Guide* provides conceptual information about actions in Foglight®, along with configuration and instructions that will help you create custom action types.

This guide is intended for Foglight cartridge developers and field engineers who need to create actions in Foglight.

- About Foglight Actions
- Getting Started
- Defining Custom Actions
- Building Target Files
- Integrating Custom Actions with Foglight
- Testing Custom Actions

About Foglight Actions

Foglight® actions are building blocks that can be bound to rules. Actions can interact with other systems, run scripts or manipulate the environment in other ways when a rule condition to which the action is bound to is met. A default Foglight installation contains a collection of core action types that you can bind to rules as required. They include email actions, command-type actions, script actions, and others. For complete information about core actions and how to add them to rule definitions, see the *Administration and Configuration Guide*.

In monitoring environments where the functionality of core Foglight actions does not meet the needs of your business requirements, you can create custom action types and add them to Foglight as required. This guide is packaged with a ZIP file, *Foglight_7.3.0_CreatingActionsFieldGuide.zip*. Use this file as a template when creating custom action types.

Getting Started

Before you get started with writing the code for custom actions, you need to ensure that your development environment includes the tools you need, create a valid directory structure, and ensure that the appropriate environment variables exist and point to the correct locations on your system. For complete information, see the following sections:

- What You Need on page 6
- Configuring Environment Variables on page 6
- Creating the Directory Structure on page 6
- Configuring Build Properties on page 7

What You Need

First thing you need to do is to ensure that your development machine has all of the required tools needed to proceed with writing and building your code. The following list identifies the required software components:

- *Java™ development environment*. You can use your existing Java environment for writing and compiling the code.
- *Apache ANT*. You will use Apache ANT for building and packaging the code. To download the latest version of ANT, visit the following Web site:

  http://www.apache.org/dist/ant/

- *Foglight® Management Server*. When you finish writing the code, you can test it by deploying the action to the server. This will require access to the Foglight installation directory and a user account for the browser interface.

Configuring Environment Variables

There are two environment variables that you need to ensure are configured as follows:

- `ANT_HOME`. Set it to point to the ANT installation on your computer.
- `FOGLIGHT_HOME`. Set it to point to the Foglight® Management Server installation on your computer.

Creating the Directory Structure

Creating a directory structure involves extracting the contents of the ZIP file, *Foglight_7.3.0_CreatingActionsFieldGuide.zip* into a local directory. The following listing illustrates the directory structure that appears after extracting the contents of the ZIP file.

Table 1. Directory structure

File/Directory	Description
build.xml	ANT build file. For a sample file listing, see Appendix: Code Samples, build.xml on page 16.
build.properties	Configurable properties file that contains build-related information about the action name, package name, and the version number of the target cartridge. For a sample file listing, see Appendix: Code Samples, build.properties on page 18.
/src	Source directory containing a sample Java™ code (see ExampleAction.java and ExampleActionMBean.java below).
/com	
/sample	
/action	
ExampleAction.java	Implementation of the MBean interface. For a sample file listing, see Appendix: Code Samples, MBean Interface Implementation on page 19.
ExampleActionMBean.java	MBean interface of the custom action. For a sample file listing, see Appendix: Code Samples, MBean Interface on page 18.

Configuring Build Properties

Configuring build properties ensures that the correct parameters are passed to the build process and that the resulting cartridge name and its version are properly set.

The *build.properties* file contains the following properties:

- `name`. Specifies the action name. It is set to `ExampleAction` by default.

  IMPORTANT: If you choose to modify this property to a different value, at a later step you will need to edit the names of the MBean interface and its implementation class (ExampleActionMBean.java and ExampleAction.java, respectively). This includes editing the file names as well as the interface/class names in *ExampleActionMBean.java* and *ExampleAction.java*, respectively. For additional information, see Defining Custom Actions on page 8.The file names use the following syntax in order to follow ANT conventions for using variables:`${implementation_class}.java${interface}Bean.java`

- `package`. Specifies the package name. It is set to `com.sample.action` in the example shown in the *Foglight_7.3.0_CreatingActionsFieldGuide.zip* file. If you intend to use a different package name and hierarchy, modify this property as required. Changing the package name requires additional modifications to the directory structure. For example, if you change the default package name to `com.mycompany.myaction`, rename the *sample* directory to *mycompany* and the *action* directory to *myaction*.

  IMPORTANT: At a later step, you will need to ensure that the correct package name is used in the package declaration at the beginning of the MBean interface and its implementation (ExampleActionMBean.java and ExampleAction.java by default). For more information about changing the package name in those files, see Defining Custom Actions on page 8.

- `version`. Specifies the version of the target cartridge.

To configure build properties:

NOTE: This procedure continues from Creating the Directory Structure on page 6.

1. Open the *build.properties* file for editing.
2. The file includes the following lines of code:

   ```
   name=ExampleAction
   package=com.sample.action
   version=1.0
   ```

   Edit the `name`, `package`, and `version` properties as required.

   For example:

   ```
   name=MyCustomAction
   package=com.mycompany.myaction
   version=2.0
   ```

   IMPORTANT: If you made changes to the package property, ensure that those changes are reflected in the directory structure. In the above example, modifying the entry `com.sample.action` to `com.mycompany.myaction` requires that you change the name of the sample directory to mycompany and the action directory to myaction.

   IMPORTANT: It is recommended that you increase the version number each time you package a new version of the action. Doing so ensures that the previous version is updated when you create and install the later version of the cartridge containing the custom action.

> **TIP:** Later on, you will need to ensure that the package name is also updated in the package declaration in the MBean interface and its implementation class. For more information, see Defining Custom Actions on page 8.

3 Save your changes and close the properties file.

Defining Custom Actions

A Foglight® action is comprised of two major components:

- MBean interface (see Writing the MBean Interface on page 8)
- MBean interface implementation (see Implementing the MBean Interface on page 9)

These action-specific components need to be saved in the `src` directory, where *package_name* follows the package hierarchy that is reflected in the directory structure. For example:

com/mycompany/someaction/SomeActionMBean.java (MBean interface)

com/mycompany/someaction/SomeAction.java (MBean interface implementation)

To find out more about the contents and structure of your working directory, see Creating the Directory Structure on page 6.

Writing the MBean Interface

An `MBean` interface is the first action-specific component that you need to write. It includes the methods that you implement at a later step

The `MBean` interface that you are about to write must meet the following requirements:

- Extends the `BaseActionMBean` class.
- Contains the following two mandatory methods:
  - `getParametersMetadata()`. A method for providing meta-data used by the browser interface to enable the end-user to supply input. This allows the user to provide the context by connecting it with a Foglight registry variable, a rule-level variable, or a custom value.
  - `invoke()`. A method that implements the action behavior.

  Depending on the nature of your custom action, the `MBean` interface typically contains additional methods that carry out your business requirements. Those methods, along with `getParametersMetadata()` and `invoke()`, will be defined in the implementation of this interface at a later step, as described in Implementing the MBean Interface on page 9.

Start by editing the sample file, *ExampleActionMBean.java*.

To write a MBean interface:

> **NOTE:** This procedure continues from Configuring Build Properties on page 7.

1 In your directory structure, locate the *ExampleActionMBean.java* file and change its name so that the file name includes the action name configured in the *build. properties* file. For more information about configuring this file, see Configuring Build Properties on page 7.

  For example, if the configured action name is `MyCustomAction`, rename the file to *MyCustomActionMBean.java.*

  For information on where to find *ExampleActionMBean.java* in the directory structure, see Creating the Directory Structure on page 6.

2 Open the file for editing.

   For a sample file listing, see MBean Interface on page 18.

3 If you previously changed the package name from its default value, `com.sample.action` while configuring *build.properties*, in the newly-renamed *.java* file, update the package declaration that appears at the beginning of the file.

   For example, if the package name is `com.mycompany.myaction`, replace the following line of code

   ```
   package com.sample.action;
   ```

   with

   ```
   package com.mycompany.myaction;
   ```

4 Edit the interface name so that it matches the file name configured in Step 1.

   For example, if the file name is *MyCustomActionMBean.java*, replace the following line of code

   ```
   public interface ExampleActionMBean extends
      BaseActionMBean
   ```

   with

   ```
   public interface MyCustomActionMBean extends
      BaseActionMBean
   ```

5 If the custom action requires any additional methods, you can specify them at this point.

6 Save your changes and close the file.

You can now proceed to write a class that implements the newly-defined `MBean` interface. For more information, see Implementing the MBean Interface on page 9.

Implementing the MBean Interface

In the implementation or the `MBean` interface you will define the behavior of all the methods that appear in the interface.

The implementation of the `MBean` interface that you are about to write must include the definitions of the mandatory `getParametersMetadata()` and `invoke()` methods, along with the definitions for any other action-specific method that appear in the interface. For more information about `getParametersMetadata()` and `invoke()`, see Writing the MBean Interface on page 8.

To implement the newly-written MBean interface:

NOTE: This procedure continues from Writing the MBean Interface on page 8.

1 In your directory structure, locate the *ExampleAction.java* file and change its name so that the file name includes the action name configured in the *build. properties* file. For more information about configuring this file, see Configuring Build Properties on page 7.

   For example, if the configured action name is `MyCustomAction`, rename the file to *MyCustomAction.java*

   For information on where to find *ExampleAction.java* in the directory structure, see Creating the Directory Structure on page 6.

2 Open the file for editing.

   For a sample file listing, see MBean Interface Implementation on page 19.

3 If you previously changed the package name from its default value, `com.sample.action` while configuring *build.properties*, in the newly-renamed *.java* file, update the package declaration that appears at the beginning of the file.

   For example, if the package name is `com.mycompany.myaction`, replace the following line of code

   ```
   package com.sample.action;
   ```

with

```
package com.mycompany.myaction;
```

4 Edit the interface and class names so that they both match the file name configured in Step 1.

For example, if the file name is *MyCustomActionMBean.java*, replace the following line of code

```
public class ExampleAction extends BaseAction implements
   ExampleActionMBean
```

with

```
public class MyCustomAction extends BaseAction implements
   MyCustomActionMBean
```

5 Implement the behavior of the mandatory `getParametersMetadata()` and `invoke()` methods, along with any action-specific other methods that are declared in the MBean interface.

6 Save your changes and close the file.

You can now proceed to build you actions and integrate them with Foglight. For more information, see Integrating Custom Actions with Foglight on page 12.

Building Target Files

The build process is comprised of a series of tasks that compile and package your Java™ code. Those tasks are described in the *build.xml* file and are executed by an ANT process. For a sample listing of the *build.xml* file, see build.xml on page 16.

To build target files:

NOTE: This procedure continues from Implementing the MBean Interface on page 9.

1 Open a Command Prompt window (Windows®) or a terminal window (UNIX® or Linux®).

2 Start the build process by issuing the following command:

Windows

```
"%ANT_HOME%"\bin\ant
```

UNIX

```
$ANT_HOME/bin/ant
```

A build log appears in the Command Prompt window or the terminal window.

Windows example

```
Buildfile: build.xml

init:

clean:

compile:
   [copy] Copying 1 file to
      C:\custom_actions\build\lib\core
   [mkdir] Created dir: C:\custom_actions\build\classes
   [javac] Compiling 2 source files to
      C:\custom_actions\build\classes

sar:
   [mkdir] Created dir: C:\custom_actions\build\sar
```

```
   [jar] Building jar:
     C:\custom_actions\build\sar\MyCustomAction.sar

cartridge:
   [unzip] Expanding:
     C:\Quest_Software\Foglight\tools\fglant.zip into
     C:\custom_actions\build\lib\ant
     [car] creating cartridge archive: C:\custom_actions\
        .\build\MyCustomAction-1_0_0.car temp file: C:\
        custom_actions\.\build\MyCustomAction-1_0_0.car2
        8653.tmp
   [cartridge] creating cartridge: MyCustomAction-1.0.0
   [cartridge] foglight version: 5.0
     [car] adding Cartridge: MyCustomAction-1.0.0
     [car] setting final flag: false on cartridge.
     [car] adding Component: MyCustomAction-sar-1.0.0
     [car] adding Item: MyCustomAction.sar
     [car] Cartridge Archive Creation Successful

example:
   [zip] Building zip: C:\custom_actions\build\example.zip

dist:

BUILD SUCCESSFUL
Total time: 4 seconds
```

The build process compiles your Java code and creates a *build* sub-directory in your directory structure. In that directory, you will find a cartridge file (.car) that contains the custom action. The CAR file is a packaging artifact that you will use to integrate the custom action. It is located at the root of the *build* directory.

The following table illustrates the structure the *build* directory and provides additional information about the directory contents where necessary.

Table 2. build directory contents

File/Directory	Description
/build	Contains the deliverable CAR file along with some temporary build files.
<action_name>-<version_number>.car	This cartridge file is the final deliverable that you can use to integrate your custom action with Foglight®. The file name uses the following syntax conventions: • *action_name* is the action name configured in *build.properties*. • *version_number* is the cartridge version number configured in *build.properties*. For example: *MyCustomAction-1_0_0.car*. For more information about the settings in the *build.properties* file, see Configuring Build Properties on page 7.
example.zip	A ZIP file containing the artifacts that can be used to build the resulting custom action and cartridge components. You can use it as a template when creating custom actions at a later time.
/classes	These directories contain the compiled Java code.

You have successful compiled and packaged the custom action code. From here, you can now proceed to integrate your custom action with Foglight. For complete information, see Integrating Custom Actions with Foglight on page 12.

Integrating Custom Actions with Foglight

The build process produces the cartridge file (CAR) that you can use to integrate the custom action with Foglight®. The cartridge file can be installed and managed in Foglight as any other cartridge file. You can install and enable the cartridge either through the browser interface or the command line using `fglcmd`'s `cartridge:install fglcmd` command. For details, see Installing CAR Files on page 12.

Installing CAR Files

A cartridge file can be quickly installed and enabled on the Foglight® Management Server using the browser interface. Another way of installing a cartridge is through the command line, by issuing the `cartridge:install` command that is included in the `fglcmd` package. The following procedure illustrates the process of installing a cartridge file through the Administration module in the browser interface. For complete details about the `fglcmd` interface and the `cartridge:install` command, see the *Command-Line Reference Guide*. For additional information about the Administration module, see the *Administration and Configuration Guide*.

The cartridge file, *<action_name>-<version_number>.car*, can be found in the *build* directory of your development environment. For details about the contents of this directory, see Building Target Files on page 10.

To install and enable a CAR file using the browser interface:

NOTE: This procedure continues from Building Target Files on page 10.

1. Start the browser interface and log in to Foglight.
2. In the browser interface, ensure that the navigation panel is open.

   To open the navigation panel, click the right-facing arrow on the left.
3. Open the Cartridge Inventory dashboard.

   On the navigation panel, under **Dashboards**, choose **Administration > Cartridges > Cartridge Inventory**.

   The Cartridge Inventory dashboard appears in the display area, showing a list of all existing cartridges.
4. Select the cartridge file that was created during the build process.

   For information about the *build* directory and its contents, see Building Target Files on page 10.

   a. Ensure that the **File on Local Computer** option is selected and click **Browse**.

   b. In the file browser that appears, navigate to the CAR file and select it.

      The file browser closes and the Cartridge Inventory dashboard refreshes, showing the location of the selected CAR file.
5. To enable the cartridge immediately after its installation, on the Cartridge Inventory dashboard, ensure that the **Enable on install** check box is selected.

   The Cartridge Inventory dashboard refreshes, showing the newly-installed cartridge in the list of installed cartridges.

Upon a successful execution of the above steps, your custom action is integrated with Foglight. You can now proceed to test the results of your custom actions. For details, see Testing Custom Actions on page 13.

Testing Custom Actions

You can test a newly-integrated custom action by creating a new rule and binding it to the custom action. This will cause the action to be invoked each time the rule condition is met.

The recommended approach for testing custom actions is to create a simple, time-driven custom rule that is triggered every ten seconds, with its condition set to `True`. This will ensure that the rule fires every ten seconds by default. Furthermore, the rule should be bound to the custom action that you are about to test. When you save the changes to the rule, you can verify if the custom action is invoked every ten seconds as specified.

The following procedure describes the process of creating a new rule and binding it to a custom action. For complete information about rules, see the *Administration and Configuration Guide*.

To create a rule and bind it to a custom action:

NOTE: This procedure continues from Integrating Custom Actions with Foglight on page 12.

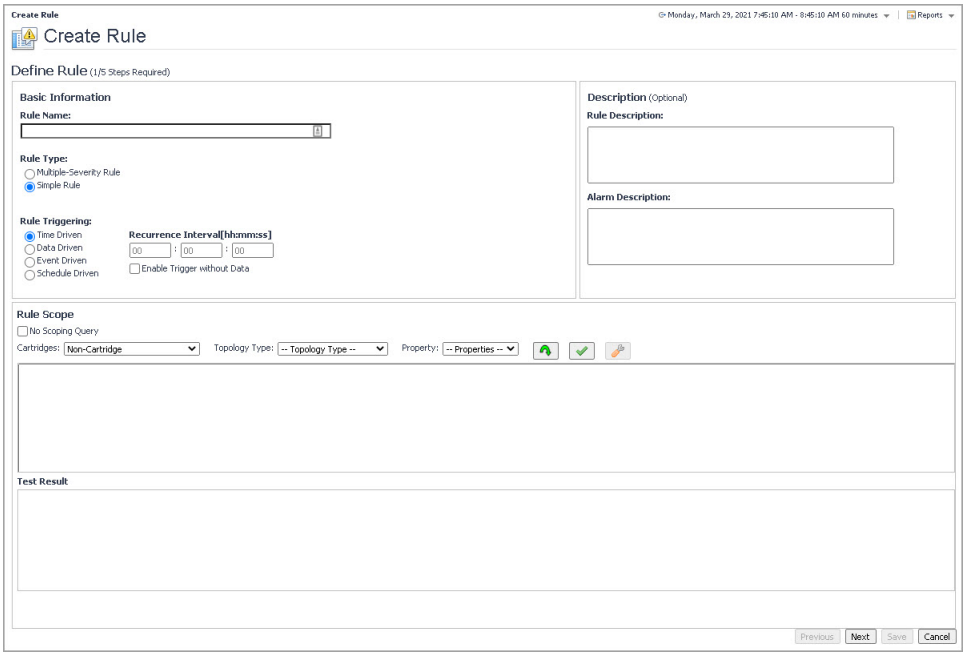
1. Start the browser interface and log in to Foglight®.
2. In the browser interface, ensure that the navigation panel is open.

   To open the navigation panel, click the right-facing arrow on the left.
3. Open the Create Rule dashboard.

   On the navigation panel, under **Dashboards**, choose **Administration > Rules & Notifications > Create Rule**.

   The rule definitions appears in the display area with the **Rule Definition** dashboard open.

   Figure 1. Rule Definition dashboard

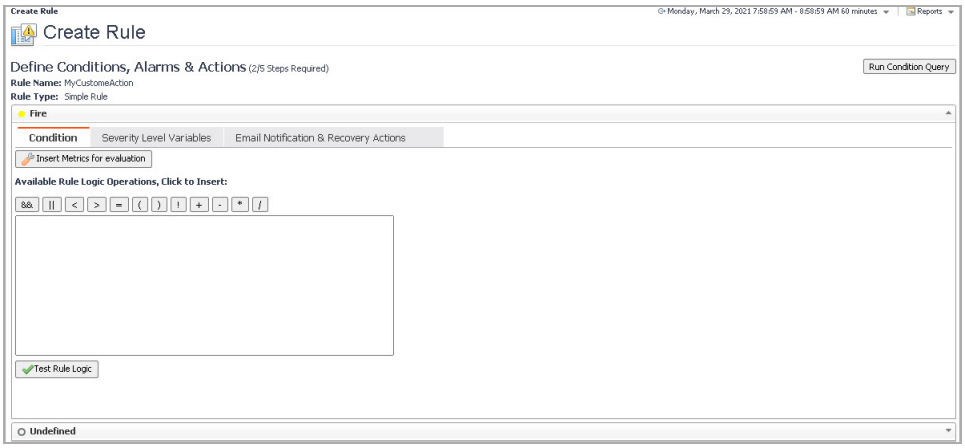


4. On the **Rule Definition** dashboard, specify the following settings:
   - **Rule Name**: Specify the rule name. For example, `MyCustomAction`.
   - **Rule Type**: **Simple Rule**.
   - **Rule Triggering**: Select the **Time Driven** option and set its **Recurrence Interval** to 10 seconds.
5. Write a rule condition consisting of a single logical expression: `True`.

a Open the **Conditions & Actions** dashboard by clicking **Next** and click **Fire** to define a condition for that state.

b Open the **Condition** tab.

Figure 2. Condition tab

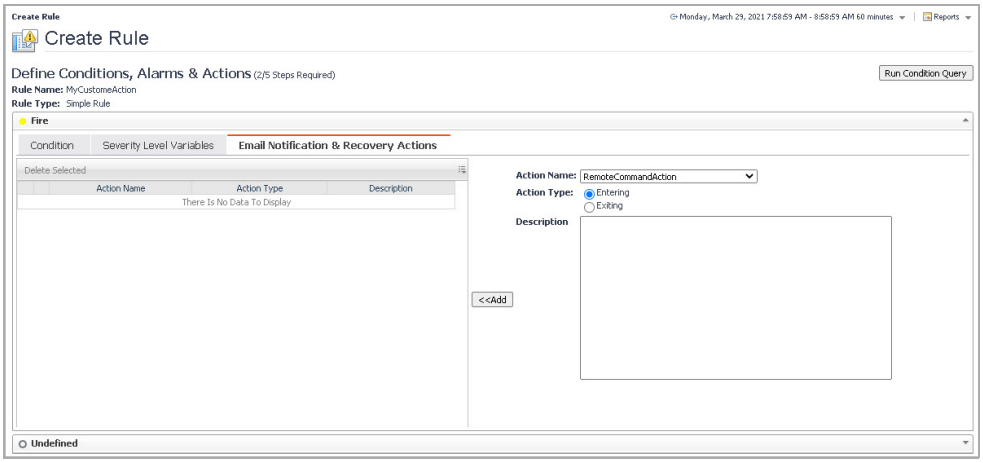


c In the **Condition** box, type `True`.

6 Bind the custom action to the rule.

a Open the **Email Notification & Recovery Actions** tab of the **Conditions & Actions** dashboard.

Figure 3. Email Notification & Recovery Actions tab

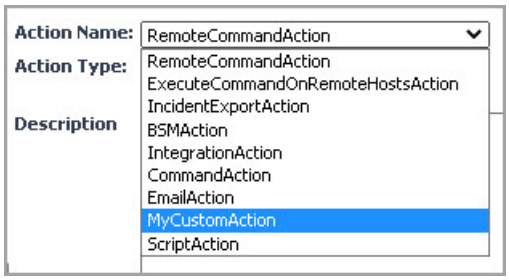


b Ensure that **Action Type** is set to **Entering**.

c Click **Action**.

Your newly defined custom action appears in the **Action** list that expands.

Figure 4. Custom action



d Select the custom action in the list and click **Add**.

The custom action appears in the **Action** table.

7 Click **Finish** to save your changes.

Upon successfully saving the rule, the newly-created rule invokes the custom action every ten seconds. The process of verifying the results of the custom action depends on the nature of the custom action.

A

Appendix: Code Samples

This appendix contains code samples that illustrate the contents of the files found in the development directory. You will find these file contents in the example.zip file packaged with this document. For information about the directory structure, see Creating the Directory Structure, Creating the Directory Structure on page 6.

- build.xml
- build.properties
- MBean Interface
- MBean Interface Implementation

build.xml

```
<project name="BuildAction" basedir="." default="dist">

   <property file="build.properties"/>
   <property name="sar" value="${name}.sar"/>

   <target name="init">

      <property environment="env"/>
      <fail unless="env.FOGLIGHT_HOME" message="Please define
         FOGLIGHT_HOME pointing to FMS installation directory"/>
      <fail unless="name" message="Please define name of action
         in build.properties"/>
      <fail unless="version" message="Please define version of
         action in build.properties"/>
      <fail unless="package" message="Please define package of
         action in build.properties"/>

      <mkdir dir="./build"/>
      <echo file="./build/implementation.properties">dir=
         ${package}</echo>
      <replaceregexp file="./build/implementation.properties"
         match="\." replace="\\/" flags="g"/>
      <property file="./build/implementation.properties"/>

      <available property="implementation" file="./src/${dir}/
         ${name}.java"/>
      <fail unless="implementation" message="Please provide
         implementation ./src/${dir}/${name}.java"/>

      <available property="interface" file="./src/${dir}/
         ${name}MBean.java"/>
      <fail unless="interface" message="Please provide interface
         ./src/${dir}/${name}MBean.java"/>

      <tstamp><format property="now" pattern="yyyy/MM/dd-
```

```
      HH:mm:ss"/></tstamp>
     <property name="buildid" value="${version}-${now}"/>

   <mkdir dir="./build"/>

</target>

<target name="cartridge" depends="init,sar">

   <!-- grab the tooling jars-->
   <unzip src="${env.FOGLIGHT_HOME}/tools/fglant.zip"
      dest="build/lib/ant"/>

   <!-- define the cartridge ant task -->
   <taskdef name="car" classname="com.quest.nitro.tools.ant.c
      artridge.Car">
      <classpath>
         <fileset dir="build/lib/ant"/>
      </classpath>
   </taskdef>

   <!-- package the car -->
   <car destdir="./build">
      <cartridge name="${name}" version="${version}"
         buildId="${buildid}">
         <component name="${name}-sar" type="Action"
            version="${version}" deploytype="DEPLOY_STAND
            ARD" deploymentitem="${sar}">
            <fileset file="./build/sar/${sar}"/>
         </component>
      </cartridge>
   </car>

</target>

<target name="compile" depends="init">

   <!-- prepare foglight.jar -->
   <copy overwrite="false" tofile="build/lib/core/foglight
      .jar" file="${env.FOGLIGHT_HOME}/server/default/deploy/
      foglight.sar"/>

   <!-- compile classes -->
   <mkdir dir="./build/classes"/>
   <javac srcdir="./src" destdir="./build/classes">
      <classpath id="car.task.classpath">
         <fileset dir="${env.FOGLIGHT_HOME}">
            <include name="lib/*.jar"/>
            <include name="server/default/lib/*.jar"/>
            <include name="server/default/deploy/
               foglight.sar"/>
         </fileset>
         <pathelement path="build/lib/core/foglight.jar"/>
      </classpath>
   </javac>
</target>

<target name="dist" depends="init,clean,cartridge,example">
 </target>

<target name="clean">
```

```
      <delete includeemptydirs="true" failonerror="false">
         <fileset dir="./build" excludes="eclipse/**"/>
      </delete>
   </target>

   <target name="sar" depends="init,compile">

      <!-- create build directory -->
      <mkdir dir="./build/sar"/>

      <!-- prepare service descriptor -->
      <echo file="./build/sar/jboss-service.xml"><![CDATA[<?xml
         version="1.0" encoding="UTF-8"?>
         <!DOCTYPE server PUBLIC "-//JBoss//DTD MBean Service
            3.2//
            EN" "http://www.jboss.org/j2ee/dtd/jboss-service
               _3_2.dtd">
         <server>
            <mbean code="${package}.${name}" name="com.quest.
               nitro.action:type=action,name=${name}"/>
         </server>
      ]]></echo>

      <!-- package sar file -->
      <jar jarfile="./build/sar/${sar}">
         <metainf file="./build/sar/jboss-service.xml"/>
         <fileset dir="./build/classes"/>
      </jar>
    </target>

   <target name="example">
      <zip destfile="build/example.zip">
         <fileset dir="." includes="*" excludes="\.*,build"/>
         <fileset dir="." includes="src/**"/>
      </zip>
   </target>

</project>
```

build.properties

```
name=ExampleAction
package=com.sample.action
version=1.0
```

MBean Interface

```
package com.sample.action;

import java.util.Collection;
import java.util.Map;
import com.quest.nitro.service.action.api.
   ActionInvocationException;
import com.quest.nitro.service.action.api.ActionParameter;
import com.quest.nitro.service.action.api.
```

```
   ActionParameterMetaData;
import com.quest.nitro.service.action.api.
   BaseActionMBean;

/**
 * Sample action MBean interface
 */
public interface ExampleActionMBean extends BaseActionMBean
{
   /**
   * action invocation - part of the BaseActionMBean contract
   */
   void invoke (Map<String, ActionParameter> parameters) throws
      ActionInvocationException;

   /**
   * action meta information - part of the BaseActionMBean
   * contract/
   Collection<ActionParameterMetaData> getParametersMetadata ();

}
```

MBean Interface Implementation

```
package com.sample.action;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Map;

import com.quest.nitro.service.action.api.
   ActionInvocationException;
import com.quest.nitro.service.action.api.
   ActionParameter;
import com.quest.nitro.service.action.api.
   ActionParameterMetaData;
import com.quest.nitro.service.action.api.BaseAction;
import com.quest.nitro.service.action.api.
   SimpleActionParameterMetaData;

/**
 * Action MBean implementation for flushing JDBC connection pools
 */
public class ExampleAction extends BaseAction implements ExampleActionMBean
{

   /**
   * Implementation for providing meta data
   /
   @Override
   public Collection<ActionParameterMetaData>
      getParametersMetadata ()
   {
      Collection<ActionParameterMetaData> result = new
         ArrayList<ActionParameterMetaData>();
      ActionParameterMetaData log = new
         SimpleActionParameterMetaData("Input", Boolean.class,
         "Whether or not", false, Boolean.TRUE);
```

```
      result.add(log);
      return result;
   }

   /**
   * Implementation for action invocation
   */
   @Override
   public void invoke (Map<String,ActionParameter> parameters)
      throws ActionInvocationException
   {
      System.out.println("Doing it - "+parameters.
         get("Input").getValue());
   }

}
```

About Us

Quest creates software solutions that make the benefits of new technology real in an increasingly complex IT landscape. From database and systems management, to Active Directory and Office 365 management, and cyber security resilience, Quest helps customers solve their next IT challenge now. Around the globe, more than 130,000 companies and 95% of the Fortune 500 count on Quest to deliver proactive management and monitoring for the next enterprise initiative, find the next solution for complex Microsoft challenges and stay ahead of the next threat. Quest Software. Where next meets now. For more information, visit https://www.quest.com/.

Technical support resources

Technical support is available to Quest customers with a valid maintenance contract and customers who have trial versions. You can access the Quest Support Portal at https://support.quest.com.

The Support Portal provides self-help tools you can use to solve problems quickly and independently, 24 hours a day, 365 days a year. The Support Portal enables you to:

- Submit and manage a Service Request.
- View Knowledge Base articles.
- Sign up for product notifications.
- Download software and technical documentation.
- View how-to-videos.
- Engage in community discussions.
- Chat with support engineers online.
- View services to assist you with your product.