WebSphere WebSphere Dashboard Framework

Version 6.0
Windows 2000 and Windows XP

Alerts Module Technical Reference

SC23-5922-00

WebSphere® WebSphere Dashboard Framework

Version 6.0
Windows 2000 and Windows XP

Alerts Module Technical Reference

SC23-5922-00

Note
Before using this information and the product it supports, read the information in "Notices" on page 27.

First Edition (March 2007)

This edition applies to version 6.0 of IBM WebSphere Dashboard Framework (product number L-JNEN-6QKLT6) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Introduction to the Alerts Module Technical Reference

Experienced developers will find in-depth technical information in this reference, including information about creating alert evaluators and creating data portlets to support alerts.

If you are new to WebSphere Dashboard Framework or are looking for basic information about alerts management and development, start with these sources of information first:

1. **WebSphere Dashboard Framework Help** - See ″Working with alerts″ for information about tasks related to defining, scheduling, and displaying alerts in an application.
2. The **Alerts Module Examples feature set** contains a sample Sales Orders portlet that leverages alerting features.
3. The **WebSphere Dashboard Framework sample application and Alerts tutorial** also demonstrate how to incorporate alerts into applications.
4. **Portlet help topics** - Both the Manage Alerts and My Alerts portlets provide help for completing tasks supported by each portlet. Portlet Help is accessible by clicking the '?' on the portlet title bar.
5. **Online documentation** is available on the IBM Web site. Select **Start → All Programs → IBM WebSphere → Dashboard Framework → Online Documentation** to link to the Web site.

Developing alert evaluators

This section describes how to develop a variety of alert evaluators that contain all of the business logic that defines when alerts are activated, their priority, and many other attributes.

Common themes in evaluator development

The actual implementation of each evaluator type involves different technology and approaches, but there are some common themes that unify the development of all evaluators.

By design all alert evaluators look and behave the same from the perspective of the alerts engine. The only apparent difference is in how the alerts engine invokes the evaluator to obtain alert instances. The themes shown below will help you extend your knowledge of one type of evaluator to build other types of evaluators.

All evaluators have an evaluate method

When the alerts engine invokes an evaluator, it calls the evaluate method. The form of this method differs but it must always be present. For example, a class-based evaluator will have a public method named ″evaluate″ while a model-based evaluator will have an evaluate method built into the model.

The evaluate method always takes the same parameters

When the evaluate method is called, the alerts engine always passes the same set of parameters. Those parameters are detailed in the table below along.

Name	Type	Description
Alert ID	String	Unique string ID of the alert definition that references the evaluator. Since a single evaluator can be referenced by multiple definitions, this gives the evaluator a way of knowing the ID of the alert definition for which it is being invoked. The evaluator may then use this ID to customize its behavior or return different sets of alerts.
Category ID	String	String ID of the alert definition's category. Evaluators will use this ID to lookup the user visible Display Category that should be assigned to any alert instances created by this invocation of the evaluate method.

Name	Type	Description
Locale	java.util.Locale	The preferred locale of the user on whose behalf alerts are being evaluated. The evaluator will typically use this parameter to identify the resource bundle to be used when looking up user visible strings such as Display Category, Display Name, and Display Text for alert instances.
Map	java.util.Map	The set of runtime properties to be used by the evaluator when it creates alert instances for this invocation. The property values are a combination of those default values specified in the alert definition and those custom values specified by the user on whose behalf alerts are being evaluated. These properties will typically be threshold values used by the evaluator to determine alert fields such as Active and Priority.

The evaluate method always returns a list of Alert instances

Each invocation of the evaluate method may return zero or more alert instances. To support this multiplicity the method must return those instances wrapped in a java.util.List object for class- and model-based evaluators. External evaluators return the list of instances encoded in XML as defined by the alerting.xsd schema in WEB-INF/solutions/alerting/schemas.

If no alert instances are generated, then the evaluator must return an empty list.

Evaluators are responsible for creating localized alert instances

When the alerts engine invokes an evaluator it passes in the preferred locale of the user on whose behalf alerts are being evaluated. The evaluator must use the locale parameter to select an appropriate resource bundle from which localized user visible strings are retrieved.

Alert instances caching is controlled by evaluators

The alerts engine maintains the cache of evaluated alert instances, but it is the evaluators that control whether their created alert instances are cached or not. Each created alert instance contains a Creation Date and an Expiration Date field. If the evaluator sets both fields, then the alerts engine caches the instances until the Expiration Date passes. On the other hand, if either field is set as null, then the alerts engine will not cache the instance. This gives evaluators the ability to control when their created alert instances are cached and for how long.

Type-specific evaluator requirements

There are three types of alert definitions: script-based, custom, and external. Custom evaluators are grouped into two types: class-based and model-based. Each type of evaluator has specific requirements.

Class-based evaluators

1. The evaluator class must be declared public and be derived from interface AlertEvaluator in the package com.bowstreet.solutions.alerting.impl. This interface defines the method named evaluate that will be invoked by the alerts engine when it evaluates the state of alerts.
2. The class must define a public no-argument constructor. This constructor will be used by the alerts engine to create instances of the evaluator class.
3. The evaluate method must return a list of zero more object instances derived GenericAlert in the package com.bowstreet.solutions.alerting.impl.
4. When deployed the class must be on the Factory's classpath so it can be loaded by the alerts engine at runtime. The Factory's classpath includes /WEB-INF/work/classes, /WEB-INF/work/lib, /WEB-INF/lib, /WEB-INF/classes, and the application server's classpath. If the evaluator class is not located in one of these places, then evaluation of the alerts associated with the definition will fail at runtime. The alerts engine will create an instance of the class each time it needs to evaluate alerts. If the class cannot be loaded or it does not implement the AlertEvaluator interface, then alerts for the associated definition will not be evaluated and the alerts engine will log an error.

Model-based evaluators

1. The model must define a public method named ″evaluate″ that returns an instance of java.util.List. Furthermore, the list must contain zero more object instances derived from com.bowstreet.solutions.alerting.impl.GenericAlert.
2. The evaluate method must take four parameters: an Alert ID, a Category ID, a locale, and a set of properties. Both IDs are java.util.Strings, the locale is of type java.util.Locale, and the properties are an instance of java.util.Map. The alerts engine will create an instance of the model and invoke the method when it needs the model to evaluate alerts associated with the definition. If a method of the specified signature cannot be found in the model, then alerts for the definition cannot be evaluated and the alerts engine will log an error.

Understanding alert instance content

This topic explains what is contained in an alert instance.

All evaluators, when invoked, will create zero or more alert instances that are returned to the alerts engine. Before any instances are created, an evaluator will typically collect application-specific business data that it uses to determine how many alerts should be generated and determine the actual field values of the created instances.

An alert instance is composed of a set of fields with specific semantic meanings. Each evaluator is required to create instances that conform to these semantics. Class- and model-based evaluators are required to return a list of Java object instances derived from the class GenericAlert in package com.bowstreet.solutions.alerting.impl.

Regardless of how an alert instance is created, its fields must obey the semantic requirements outlined in the following section.

Alert field semantics

The fields in an alert instance must obey the following semantic requirements.

Alert Field Name	Required	Description / Semantic Meaning
Alert ID	Yes	Unique string ID of this alert instance that is derived from the ID of the alert definition used to generate the instance. It serves as a way for the alerts engine to trace an instance back to its associated definition.
Category ID	Yes	String ID of the category to which this instance belongs. Applications will use this ID to lookup the user visible name for the category.
Creation Date	Yes	Positive long value indicating the date and time at which the logical alert represented by this instance was created. This is not a timestamp of the instance's creation.
Detailed Display Text	No	The alert's detailed text in a form that is displayable to a user. The evaluator must set the text using the preferred locale of the user requesting the alert.
Display Category	No	The alert's category in a form that is displayable to a user. The evaluator must set the text using the preferred locale of the user requesting the alert.
Display Name	No	The alert's name in a form that is displayable to a user. The evaluator must set the text using the preferred locale of the user requesting the alert.
Display Text	No	The alert's text in a form that is displayable to a user. The evaluator must set the text using the preferred locale of the user requesting the alert. This string will typically be an abbreviated form of the Detailed Display Text field.

Alert Field Name	Required	Description / Semantic Meaning
Expiration Date	No	Positive long value indicating the date and time at which the logical alert represented by this instance should no longer be cached.
Priority	No	Integer value indicating the priority of this alert. Note: a value of 1 is higher priority than a value of 2.
Version	No	Free form string used by the evaluator to mark the version of this alert instance. The alerts engine uses this version string to determine when two or more alerts with the same ID are really different and should be treated as such for caching purposes. The version is also used by the alerts engine to determine when an alert deleted by a user should be displayed to again because it is new.

Handling special alert fields

Follow the rules in this section for alert fields to set appropriate values in an evaluator.

Setting the Alert ID field

Follow these rules when setting the Alert ID value for the evaluator.

The Alert ID of an instance is derived from the unique Alert ID of the definition from which the instance was generated. This behavior allows all alert instances to be traced back to their associated definitions and to be sorted, categorized, and filtered by portlets and the alerts engine.

The Alert ID value set by the evaluator is determined by the nature of the data the evaluator uses in its calculations and whether it is possible for the evaluator to return multiple instances of an alert for a single invocation. Here are the rules you must follow:

1. If the evaluator can only return a single alert instance for each invocation, then the Alert ID of an instance must be set to the Alert ID passed in to the evaluator by the alerts engine. This scenario is most common when the data used in the evaluation is a single value or there are multiple values, but they do not need to be represented in finer detail for alerting purposes.
2. On the other hand, if the evaluator can return multiple instances for each invocation, then the Alert ID passed in to the evaluator must be suffixed with an appropriate identifying string before being used to set the Alert ID of each instance. This scenario is very common when an evaluator deals with sets of related data and each member of the set may require an individual alert instance. A good example is an evaluator that deals with EBITA data that is broken out by company division and the business requirements dictate that an

alert should be generated for each division that falls below some threshold. Note: EBITA is a business financial term meaning Earnings Before Interest, Taxes, and Amortization.

If the evaluator chooses to add an identifying string suffix to the Alert ID, then the suffix must be separated from the Alert ID by a forward slash. The forward slash is used by the alerts engine to determine where the Alert ID ends and the suffix begins.

The identifying string must also be unique so that each alert instance returned has a unique Alert ID field. An evaluator should choose an identifying string that is meaningful to portlet developers and, if possible, derived from the set of data used to generate the set of alert instances. For example, for an evaluator that deals with EBITA data broken out by division, you could use the unique division name as the identifying string. Thus each created alert would be assigned a unique ID that traces back to both the alert definition from which is was generated and the divisional data used to set the instance's field values. One such Alert ID might look like this: ″EBITA/ConsumerProducts.″

Setting the Category ID Field

The Category ID of an instance should be set to the Category ID passed in to the evaluator by the alerts engine. This is the Category ID of the definition from which the instance is being generated.

Setting the Creation Date field

The creation date of an alert instance is the date on which the logical business alert it represents is considered to have been created in the system.

For example, a Sales alert instance that is based upon the last month's data would have a creation date set to the first day of the current month, if the expiration of its definition is ″month.″ For example, instances of an alert created using April 2007 data would have a creation date of ″01 May 2007″ to signify that the alert was logically created on the first day after the month closed. This approach ensures that alert instances accurately represent logical business semantics regardless of when they are physically created by an evaluator.

Setting the Expiration Date Field

The Expiration Date of an alert is the date upon which the alert expires and is no longer cached.

The alerts engine uses the alert's Expiration Date to determine when the alert is stale and needs to be re-evaluated. The alerts engine considers an alert stale and re-evaluates it when Expiration Date is less than the current system time. For example, a Sales alert created in April 2007 would calculate an Expiration Date of one month after the Creation Date since its state is valid for the entire month of May 2007. That causes the alerts engine to cache the instance for a month. When June 2007 arrives, the alert for May 2007 becomes stale and the alerts engine will again call the evaluator. This time the evaluator will return an alert instance with a Creation Date of ″01 June 2007″ and an Expiration Date one month later. In this way the evaluator can control the caching behavior of the alerts engine.

Setting the Version field

The Version string is a free form field used by evaluators to encode the version of the alert represented by a specific alert instance.

The importance and use of the Version field is best illustrated by an example.

Consider an EBITA alert evaluator that generates a set of new alert instances on the first day after a quarter closes. The alert instances will have the same Alert ID values as those generated for the previous quarter (because they were generated from the same alert definition). Since the alerts engine has no knowledge of how the instances are created, it is unable to distinguish between a set of old instances and a new set. The Version field is used tell the alerts engine that an instance is new and, more importantly, that it should be presented to portlet users as a new instance even if they DELETED the previous instance of the same Alert ID.

The evaluator should set the Version field so that it changes in lock step with the data used to generate alerts. For example, if the evaluator uses quarterly data then it could use a string such as ″2Q 2007″ to signify that the instance is associated with data from the 2nd quarter of 2007.

Creating a simple class-based alert evaluator

This section explains how to create a simple class-based threshold evaluator.

Creating the stub evaluator class

This topic shows sample code for a stub evaluator class.

The code shown below conforms to the type-specific requirements for a class-based evaluator. Most importantly it contains a default constructor, derives from AlertEvaluator, and the evaluate method returns an empty java.util.List of alert instances.

```
package com.mycompany.alerting.evaluators;

import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.Map;

import com.bowstreet.solutions.alerting.AlertsEngineException;
import com.bowstreet.solutions.alerting.impl.AlertEvaluator;

public class ThresholdEvaluator implements AlertEvaluator
{
    public ThresholdEvaluator()
    {
        super();
    }

    public List evaluate( String alertID, String categoryID, Locale locale, Map properties )
        throws AlertsEngineException
    {
        return new ArrayList();
    }
}
```

1.
2.
3.

Creating an unpopulated alert instance and returning it in the list

The GenericAlert class is included as part of the Alerts Module features set and is to be used by evaluators for creating alert instances.

The evaluate method is modified to create an instance of ″GenericAlert″ and wrap it in the list returned to the alerts engine. The GenericAlert class is included as part of the Alerts Module features set and is to be used by evaluators for creating alert instances. Unless you need to add custom fields or behavior to an alert instance you should always just create instances of ″GenericAlert.″

```
package com.mycompany.alerting.evaluators;

import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.Properties;

import com.bowstreet.solutions.alerting.AlertsEngineException;
import com.bowstreet.solutions.alerting.impl.AlertEvaluator;
import com.bowstreet.solutions.alerting.impl.GenericAlert;

public class ThresholdEvaluator implements AlertEvaluator
{
    public ThresholdEvaluator()
    {
        super();
    }

    public List evaluate( String alertID, String categoryID, Locale locale, Map properties )
        throws AlertsEngineException
    {
        final List alerts = new ArrayList();
        final GenericAlert alert = new GenericAlert();

        alerts.add( alert );

        return alerts;
    }
}
```

Populating required fields in the alert instance

This topic shows sample code for setting properties for required fields.

Since the business logic has not been added yet, mark the alert as inactive. The Alert ID and Category ID fields are set according to the values passed in by the alerts engine. This causes the alert instance to be traceable back to the alert definition used by the alerts engine to locate and invoke this evaluator. Finally, use the current system time to set the creation date. Since the Cache Timeout field was not set this alert instance will not be cached by the alerts engine.

```
public List evaluate( String alertID, String categoryID, Locale locale, Map properties )
    throws AlertsEngineException
{
    final List alerts = new ArrayList();
    final GenericAlert alert = new GenericAlert();

    alert.setAlertID( alertID );
    alert.setCategoryID( categoryID );
    final Date currentDate = new Date();
    alert.setCreationDate( currentDate );

// re-evaluate alert after 300 seconds
    alert.setExpirationDate( new Date( currentDate.getTime() + (300 * 1000) ) );

//  set the version ID using the creation date and all of the properties to ensure uniqueness
//  and a predictable, repeatable value to make it retrievable from a database by version ID
    final String evaluatedVersionID = alert.getCreationDate().toString();
    final String calculatedVersionID = properties == null ? "[]":properties.values().toString();
    alert.setVersionID( StringUtil.strcat( evaluatedVersionID, ",", calculatedVersionID ) );
```

```
//  Add this alert to the List
    alerts.add( alert );

    return alerts;
}
```

Adding business logic to an alert evaluator

The ″alertIsActive″ method contains all of the business logic needed to calculate the state of the alert instance you created.

Following the example steps earlier, the alert instance is active only when the business value falls below some threshold. A few key points are worth highlighting.

The business value and threshold are obtained from the properties passed into the evaluator by the alerts engine. The names ThresholdEvaluator.businessValue and ThresholdEvaluator.threshold are defined as constants near the top of the class. But more importantly, these names are exactly the set of property names you will create as part of the alert definition for this evaluator. In essence, we have defined the set of properties used by a portlet to customize the evaluator's behavior at runtime on a user-by-user basis.

ThresholdEvaluator.businessValue will be provided by a portlet and represents alert-able data such as EBITA. The other property would likely appear in a portlet's customizer to allow a user to specify a custom threshold. In this way the alert definition, evaluator, and resulting alert instances are customized to specific users even though the alerts engine uses a single alert definition and evaluator for all users.

```
public static final String BUSINESS_VALUE = "ThresholdEvaluator.businessValue";
public static final String THRESHOLD = "ThresholdEvaluator.threshold";
...

protected boolean alertIsActive( Map properties )
{
    boolean isActive = false;

    if( properties != null )
    {
        try
        {
            // Fetch the evaluation values to be used with the business logic.
            final double businessValue =
                Double.parseDouble( properties.get(BUSINESS_VALUE).toString() );
            final double threshold =
                Double.parseDouble( properties.get(THRESHOLD).toString() );

            // Here is the business logic. The alert is active when the business value
            // falls below the configurable threshold.
            isActive = businessValue < threshold;
        }
        catch( NumberFormatException nfe )
        {
            // An evaluation parameter was not a valid double or was null.
            throw new AlertsEngineException( nfe );
        }
    }

    return isActive;
}

public List evaluate( String alertID, String categoryID, Locale locale, Map properties )
```

```
    throws AlertsEngineException
{
    final List alerts = new ArrayList();
    final GenericAlert alert = new GenericAlert();

    alert.setAlertID( alertID );
    alert.setCategoryID( categoryID );
//  set dates, version ID, etc.

//  add this alert to the List
    alerts.add( alert );

    return alerts;
}
```

It is also worth noting that the evaluate method has changed very little when we added the business logic. In fact, the only change is in the statement that sets the alert instances Active field; you are now calling the business logic method ″alertIsActive″ to determine the appropriate value for this field.

Setting localized text

Add some localized text to the alert instance so that it can be displayed to a user in their preferred locale.

This step involves creating resource bundles for each language the evaluator needs to support, modifying the evaluate method to get the resource bundle that best matches the locale passed in by the alerts engine, and using the bundle to fetch localized strings for the displayable fields of the alert instance.

The code changes are illustrated below. A set of constants has been added that reference the base name of the resource bundle and several property names that will appear in all bundles. The evaluate method now has code to get a resource bundle and use it to set the Displayable Name, Displayable Category, Displayable Text, and Detailed Displayable Text for the alert instance. Note that in this example the detailed text is the same as the displayable text. You could make the detailed text more informative by including the business value and threshold in the text.

```
private static final String BUNDLE_FILE = "com.mycompany.alerting.evaluators.messages";
private static final String DISPLAY_NAME = "DISPLAY_NAME";
private static final String DISPLAY_CATEGORY = "DISPLAY_CATEGORY";
private static final String DISPLAY_TEXT = "DISPLAY_TEXT";

public List evaluate( String alertID, String categoryID, Locale locale, Map properties )
    throws AlertsEngineException
{
    final List alerts = new ArrayList();
    final GenericAlert alert = new GenericAlert();

    alert.setAlertID( alertID );
    alert.setCategoryID( categoryID );
    alert.setCreationDate( new Date());

    // Get the resource bundle that best matches the user's preferred locale.
    final Locale localeForBundle = (locale != null) ? locale : Locale.getDefault();
    final ResourceBundle bundle = ResourceBundle.getBundle( BUNDLE_FILE,
       localeForBundle, this.getClass().getClassLoader() );

    // Set the alert's display name, category, text, and detailed text using
    // the fetched resource bundle.
    alert.setDisplayName( bundle.getString( DISPLAY_NAME ) );
    alert.setDisplayCategory( bundle.getString( DISPLAY_CATEGORY ) );
    alert.setDisplayText( bundle.getString( DISPLAY_TEXT ) );
    alert.setDetailedDisplayText( alert.getDisplayText() );
```

```
    alerts.add( alert );

    return alerts;
}
```

For reference here is a sample resource bundle named ″messages.properties″ that could be used with this class. Note that it defines strings for each of the three alert instance fields we want to localize. In this example, we have chosen to make this an EBITA threshold alert evaluator. To be consistent with the above example, this file's full path is ″WEB-INF/work/source/com/mycompany/alerting/evaluators/messages.properties.″

```
DISPLAY_NAME=EBITA Threshold Alert
DISPLAY_CATEGORY=EBITA
DISPLAY_TEXT=EBITA is below threshold.
```

At this point you have a fully working evaluator that implements a simple threshold algorithm. The next step is to create an alert definition that references this evaluator and defines the properties recognized by the evaluator; namely ″ThresholdEvaluator.businessValue″ and ″ThresholdEvaluator.threshold″

Creating the alert definition

Use the Manage Alerts portlet to create an alert definition.

Related tasks

Creating or editing an alert definition
Use the alert definition wizard to create or change settings for an alert definition.

Testing the new alert definition and evaluator

This topic describes how to test an alert evaluator.

The simplest way to test an evaluator is to modify a copy of one of the sample models that gets installed with the Alerts Module feature set. In the Portlet Factory Designer make a copy of the model named ″MyAlertsPortlet″ in models/solutions/alerting. Name the new model something like ″TestEBITAAlert″ and then open it for editing.

Open the builder call named ″getAlertEvalProperties.″ This method is used by the model to get the alert evaluation values that have been customized by the user. You will rewrite the method so that it sets values for the ″ThresholdEvaluator.businessValue″ and ″ThresholdEvaluator.threshold″ properties in a way that forces the alert to be active. Edit the method body so that it appears as follows. Then apply the changes and run the model.

```
{
    final Properties properties = new Properties();

    // Custom properties for the EBITAAlert sample.
    properties.setProperty( "ThresholdEvaluator.businessValue", "100.0" );
    properties.setProperty( "ThresholdEvaluator.threshold", "200.0" );

    return properties;
}
```

The business logic added to the evaluator marks the alert as active when the business value is less than the threshold. This set of properties will satisfy that logic. Running the model produces a page and an alert with the name ″EBITA

Alert″ now appears. This demonstrates that the alerts engine was able to invoke the evaluator and that the evaluate method was able to create a localized alert instance.

Creating a model-based evaluator

This section shown you how to create a model-based evaluator.

The process of creating a model-based evaluator is similar to that of a class-based evaluator. The main difference is in how the evaluator is packaged and referenced by an alert definition. You will use the class-based evaluator developed in the previous section as a starting point.

Examining the LJO's source file

The source file for the LJO, Orders_Example_MBE_LJO.java, includes evaluation logic that compares the values in each row of data in supplied by the data service provider against some threshold values. Each data row meeting the criteria causes an alert to be generated.

Creating the evaluator model

Create a new empty model that contains a Linked Java Object builder and an Action List builder.

Follow these steps to create an evaluator model in the same project that you used for the simple class-based evaluator.

1. In the Portlet Factory Designer, create a model named ″ThresholdEvaluator.″
2. Add a Linked Java Object builder named ″thresholdEvaluator.″ Use the Class Name picker to select the class name com.bowstreet.solutions.alerting.examples.Orders_Example_MBE_LJO. Then save the builder call.
3. Now add an Action List builder call to the model and set its name, parameters, and return type to be the same as the evaluate method you defined in the ThresholdEvaluator class.

   All evaluators must have an evaluate method that takes the same parameters and returns values of the same type. You are creating a WebApp evaluate method that will be a proxy for the actual evaluate method you developed for the simple class-based evaluator example described earlier. The Arguments section of the Action List builder call window should have the argument names and data types shown in the following table.

Argument Name	Data Type
alertID	String
categoryID	String
locale	java.util.Locale
defaultParameters	java.util.Map

4. Set the Return Type to be java.util.List.
5. Add actions to the list that invoke the setter methods of the Linked Java Object's Java class by clicking the chooser at the far right side of each row in the Actions table.

Set the AlertID, Category, Locale, and DefaultParameters. Pass the corresponding arguments to each method. Finally, use a Return action to get the alerts.

The Action List should show the following actions when you are done:

```
EvaluatorLJO.setAlertID(${Arguments/alertID})
EvaluatorLJO.setCategoryID(${Arguments/categoryID})
EvaluatorLJO.setLocale(${Arguments/locale})
EvaluatorLJO.setDefaultParameters(${Arguments/defaultParameters})
Return!${MethodCall/EvaluatorLJO.getAlerts}
```

6. Save the builder call.

Examining the XML of the sample alert model-based evaluator

The example model-based evaluator works with the Orders_Example_MBE alert definition.

The example application portlet called ″Alerts Module Example - Sales Orders Portlet″ uses a data service provider that is referenced in the XML of the alert definition. You cannot see this reference when you are working with the alert definition, but if you open the alert definition's XML file, you will see the DataSource reference. This XML file can be found at the following location in the project:

WEB-INF/solutions/alerting/xml_persistence/alert_defs/examples\
Orders_Example_MBE.xml

The following XML is the key element in the alert definition that associates this definition with a service provider. Without this reference, the model-based evaluator would not be able to generate alerts for the data in the Sales Orders Portlet.

```
<DataProviders>
<DataProvider>
<Model>solutions/alerting/examples/data_providers/Alerting_Examples_Provider_Orders_Stub</Model>
<FetchDataMethod>getAllRecordsReturnSampleData</FetchDataMethod>
<FetchDataFromRowMethod>getAlertPropertiesForRow_ordersAlertData</FetchDataFromRowMethod>
</DataProvider>
</DataProviders>
```

The model named Alerting_Examples_Provider_Orders_Stub is the data service provider model for the Sales Orders portlet that comes with the Alerts Module Examples feature set.

Related tasks

Creating or editing an alert definition
Use the alert definition wizard to create or change settings for an alert definition.

Adding alerts to a portlet

This section covers the high-level steps of creating an alert and adding it to both a data portlet and to a My Alerts portlet.

A data portlet is any portlet where specific alerts are evaluated and the results are displayed or used to drive formatting of page content. The My Alerts portlet displays a summary of all of the alert instances a user is allowed to see.

Preparing to add alerts to a portlet

Before you can add alerts to a portlet, set up the data service providers and the alert definition.

Creating data service providers

The recommended approach for handling data access (especially access to data required by alert evaluators)- is to centralize the access into one or more Service Provider models. These models then become the one-stop resource in your application for retrieving application data no matter who needs it.

Successful alert development and integration depends upon the careful design and implementation of access to an application's business data. An alert evaluator will typically need to access several pieces of application data when it is invoked by the alerts engine to generate alerts for a portal user. A data portlet that uses alerting to drive the display of fields can often pass some or all of the required data into an evaluator since the portlet will have already fetched the data for display to a user. However, when an evaluator is invoked to provide alerts for display on a My Alerts portlet, it is the responsibility of the evaluator to fetch the required data. Creating an implementation that ensures business data is available to both a data portlet My Alerts portlet is the central challenge of alert integration.

When an application's portlets are being designed careful thought must be given to whether or not the portlets will ever need to use alerting functionality. Trying to integrate alerting functionality after-the-fact can be difficult especially in cases where alert evaluators need large amounts of data and the application's data access functionality is mixed into the logic of the portlets or spread across multiple portlets of an application. A safe approach is to assume that your portlets will require access to alerting functionality and plan accordingly.

Using Service Provider models has several important benefits.

1. Changes or bug fixes to an application's data access code are applied in one place and they immediately become available to all portlets and alert evaluators that use the Service Provider.
2. The application can use caching inside the Service Provider to improve runtime data access performance of portlets and alert evaluators.
3. Portlets and alert evaluators will get consistent results when accessing the same business data.

   More precisely, since data access is centralized the application developer can guarantee that the same code is used to access business data and therefore the same values will be returned (taking into account changes effected by cache timeout and data changes in back-end systems).

4. Access control, logging, and auditing are easier to implement and manage since there is only one place where portlets can go to get data.

Defining alert evaluation properties

When designing an alert evaluator, give careful thought to the values you want users to be able to customize at runtime. These values will typically be thresholds that control when an alert is activated.

For example, consider the following example for an EBITA alert evaluator that determines when the alert is active.

```
// The Alert is active when the actual EBITA is less than our desired threshold.
if EBITA-Actual < EBITA-Threshold then Activate-EBITA-Alert
```

The EBITA-Threshold value referenced above is a good candidate to represent as an alert evaluation property. By making it a property you give users the ability to set custom threshold values and thus create customized alerts.

The EBITA-Actual value is an example of application business data that needs to be accessed by the alert evaluator at runtime. A data portlet can provide this value or the evaluator can fetch the data itself. If the evaluator expects the portlet to provide this data, then an EBITA-Actual property must be defined as well.

Note: When defining an alert evaluation property, also provide a default value. The evaluator uses this default value if one is not provided by a portlet at runtime. In general you should select default property values so that the alert evaluator will function reasonably well even if a portlet does not provide any of the required values. The recommended strategy is to select default values so that the alert is evaluated as being inactive and any other calculations complete normally. If an evaluator throws an exception when it is invoked (for example attempting to divide by zero) then the alerts engine will log the error and skip over the evaluator. However, it is considered very poor design to use the alerts engine exception handling to cover up deficiencies in an evaluator's implementation.

Implementing the alert evaluator

This section explains how to enable the evaluator to determine whether it has been invoked to support a Data portlet or a My Alerts portlet. This step is important because it will enable your alerts to work correctly in the context of a Data portlet or a My Alerts portlet.

By convention the java.util.Map instance passed to an alert evaluator may contain a string-valued property named ″evaluationMode.″ When this property is present and has a value of ″single″ the evaluator is being told by the alerts engine to only evaluate a single instance of the alert. If the property is not defined or has any other value, then the evaluator is free to evaluate as many alert instances as needed based upon the business data it is designed to access. A concrete example will make this clear.

Example

Consider an alert definition and evaluator that deals with sales results for a company. The sales data is broken out by region and in one data portlet the sales data for each region is displayed as rows in a table; one region per row. This portlet has been designed to alert the user when sales for a region goes below some configurable threshold. If the sales data in a row is below the threshold then

a sales alert for that region is considered active and the row is highlighted in red. The portlet contains an Alert Evaluation builder that is responsible for evaluating the sales alert for each row and applying appropriate formatting based upon the resulting alert's state. Before each row of sales data is rendered the Alert Evaluation builder goes into action gathering up sales data from the row and passing it to the alerts engine as a set of ″java.util.Map.″ These alert evaluation properties are then passed to the sales alert evaluator that uses the data to evaluate the state of the sales alert for the region represented by the row.

For each row of sales data the Alert Evaluation builder also adds evaluationMode with a value of ″single″ to the property set passed to the evaluator. In effect the builder is telling the evaluator it is being invoked for a data portlet and that only a single alert should be evaluated. The alert returned by the evaluator is then used by the builder to set the row's formatting.

On a My Alerts portlet the user needs to see a summary of all the currently active alerts, including any sales alerts. This portlet uses the alerts engine builder to obtain all active alerts visible to the user. When the sales alert evaluator is invoked it does not find the evaluationMode property and therefore it knows that it must obtain the current sales data itself and generate as many sales alerts as necessary. The evaluator uses a Service Provider model to obtain the sales data broken out by region. The evaluator iterates over the sales data and evaluates the state of the alert for each region. The set of alerts is then returned to the alerts engine which passes the alerts to the builder for display in the My Alerts portlet.

The following code fragment illustrates how the body of the evaluator's ″evaluate″ method could be structured to handle the evaluationMode property.

```
try
{
  List alerts = new ArrayList();

  // See if the alert is being evaluated in the "single" mode. This
  // mode indicates that iteration is controlled by the caller and the
  // method should only evaluate a single instance of the alert.
  final boolean isIteratedEvaluation = "single".equals( properties.getProperty( "evaluationMode" )

  if( isIteratedEvaluation )
  {
    // The calling portlet is controlling iteration so we are being
    // invoked for a data portlet. Just perform a single evaluation
    // and return a single alert instance.
  }
  else
  {
    // We are controlling iteration so we are being invokes for a My Alerts
    // portlet. Get all of the required business data and return a list of
    // appropriate alerts. As an alternative, we could also just return a
    // single summary alert for display in the My Alerts portlet.
  }

  return alerts;
}
catch( AlertsEngineException aee )
{
  throw new WebAppRuntimeException( aee );
}
```

Enabling the Alert definition

When development of the alert evaluator is complete, use the Manage Alerts portlet to enable the alert definition so that the alerts engine will begin invoking the evaluator when portlets make alert requests.

Creating a data portlet

Create a data portlet to flag data as alertable.

Design guidelines for data portlets

After you have a basic data portlet working, follow these guidelines to achieve consistency for alerts.

- Use the Service Provider models created to support data access from portlets and evaluators.

  This will ensure that all portlets and evaluators are getting a consistent set of data.
- Create a stylesheet that declares the formats you need when using alerts to drive page formatting.

  The portlet will most likely already employ a common stylesheet used to set formatting of page elements across your application. You can add to this common stylesheet or you can create a separate stylesheet that only declares formats used with alerts. Whichever approach you take, there should be one place for all alert-based formatting declarations.

Adding an Alert Data builder to the Service Provider model

The Alert Data builder is the key element to getting the alerts engine working with a Service Provider model.

The alerts engine needs to be able to retrieve data from the Service Provider model. It also needs to know which data is evaluated for alerts. The alerts engine can use this data to compare against threshold values. When the data is out of range, the alerts engine generates an alert.

In an Alert Data builder call, Table-Based Parameters can be declared. Those parameters declared in the Table-Based Parameters section correspond with field names in the data. Only these fields can be examined by the alerts engine while performing the computations to determine if an alert should be generated.

This builder call also has Discrete Parameters. These parameters, when declared, are most often used as the default threshold values. These thresholds are compared against the actual data using the logic implemented in the alert evaluator to determine if an alertable condition exists regarding any given row of data.

Adding a Status Indicator builder to the data portlet model

The Status Indicator builder applies styles defined in a data portlet to provide visual cues to the users when alerts have been generated.

The Alerts Module Example - Sales Orders portlet demonstrates the use of the Status Indicator builder. The portlet contains two different Status Indicator builders that cause certain visual cues to appear in the portlet.

Related concepts

Status Indicator builder
Use this builder to automate the highlighting and styling of data values to show status or alerts based on some logic.

Creating a My Alerts portlet

The Alerts Module feature set comes with a My Alerts portlet, but you can also create your own.

A My Alerts portlet traditionally displays a summary of all of the alerts a user is allowed to see. This summary is typically displayed as a table where each row represents a single alert and, optionally, one or more columns offer links to an alert details page. This is the standard master/detail pattern implemented in many of the higher-level view & form builders in Portlet Factory. The first step in creating a My Alerts portlet is to get a basic master/detail portlet built and working.

Next, add a single instance of the alerts engine builder to the portlet to obtain access to alerts engine functionality. This builder adds an LJO to the portlet that contains several methods for querying the alerts engine for alerts. The builder's inputs are used to tell the alerts engine about the user on whose behalf alerts are being requested. Care should be taken to use the same inputs so that the alerts engine returns data consistently for a user when accessed from a data portlet or a My Alerts portlet.

The alerts engine builder also has two inputs that are used to feed properties to the alert evaluators. They are under the Alert Properties group and taken together are functionally equivalent to the Evaluation Value input from the Alert Evaluation builder. The Property Source input of the alerts engine builder specifies how evaluation properties are to be defined: either as an indirect reference or explicitly as a builder input. Either way, these inputs are used to pass customized thresholds to the alert evaluators so that they can return alert instances customized for a specific user.

The list of alerts that should be displayed to a user is obtained by calling the getvisibleActiveAlerts() method on the alerts engine LJO. The list is returned as a java.util.List instance containing zero or more alerts objects derived from the class Alert in the com.bowstreet.solutions.alerting package. You can use the Bean Master Detail builder to display the list of alerts.

1. Access to alerts engine functionality is obtained by adding a single instance of the alerts engine builder to the portlet. This builder adds an LJO to the portlet that contains several methods for querying the alerts engine for alerts.
2. As with the data portlet, the builder's inputs are used to tell the alerts engine about the portlet user on whose behalf alerts are being requested. Care should be taken to use the same inputs so that the alerts engine returns data consistently for a user when accessed from a data portlet or a My Alerts portlet.
3. The alerts engine builder also has two inputs that are used to feed properties to the alert evaluators. They are under the Alert Properties group and taken together are functionally equivalent to the Evaluation Value input from the Alert Evaluation builder. The Property Source input of the alerts engine builder specifies how evaluation properties are to be defined: either as an indirect reference or explicitly as a builder input. Either way, these inputs are used to pass customized thresholds to the alert evaluators so that they can return alert instances customized for a portlet user.

4. The list of alerts that should be displayed to a user is obtained by calling the getvisibleActiveAlerts() method on the alerts engine LJO. The list is returned as a java.util.List instance containing zero or more alerts objects derived from the class Alert in the com.bowstreet.solutions.alerting package. You can use the Bean Master Detail builder to display the list of alerts.

Alerts directories

This topic lists the location of the installed Alerts Module and Alerts Examples feature sets on your workstation.

After installation of the Alerts Module and Examples feature sets, the WebApp project will contain an implementation of the alerts engine, several administration models, and several sample alert definitions. The following table summarizes the major groups of artifacts that are installed as part of this feature set. All of the paths in the table are relative to the WEB-INF directory in the WebApp project.

Path	Description
models/solutions/alerting/admin	Contains models used to create, edit, and delete alert definitions. Contains portlet models for administrators and system analysts to manage alert definitions. Also contains the My Alerts portlet used by end users to manage the alerts they receive.
models/solutions/alerting/examples	Contains sample models demonstrating how to integrate portlets with the alerts engine. Contains sample code demonstrating how to write the various types Java classes that can be implemented for use with the alerts engine. Included here are example code for a class-based evaluator, a LJO for a model-based evaluator, a custom notifier, and a custom escalation handler.
models/solutions/alerting/examples/evaluators	Contains a sample used to demonstrate writing a model-based alert evaluator.
solutions/alerting/config	Contains configuration files needed by the alerts engine. These files include a properties file containing settings for alert escalation handlers.
solutions/alerting/db_persistence	Contains additional folders with configuration and setup files needed to employ the database persistence manager in conjunction with a relational database.
solutions/alerting/schemas	Contains XML schemas defining the allowable structure and content of various XML-based alerts definitions. The alert definitions contained in the alert_defs folder must conform to these schemas. Also contains schemas defining the structure and content of XML passed to and from external alerts.

Path	Description
solutions/alerting/xml_persistence	This directory contains the storage locations for the various types of data used by the alerts engine. The XML files contained in the subdirectories in the Designer represent default data. At runtime, data created or modified through the alerts module's portlets will be stored in these same directories in the deployed applications folder on the portal server.
solutions/alerting/xml_persistence/ acknowledgements	This directory contains alert acknowledgements persisted by the alerts engine when the engine has been configured to use an XML-file persistence manager.
solutions/alerting/xml_persistence/ alert_defs	This directory contains alert definitions persisted by the alerts engine when the engine has been configured to use an XML-file persistence manager.
solutions/alerting/xml_persistence/ alert_defs/examples	Contains all of the sample alert definitions. These definitions are stored as XML files and can be used with the XML-file pPersistence manager. These sample definitions include sample alert definitions for model-based, java-based, and external alerts.
solutions/alerting/xml_persistence/ generic_alerts	This directory contains alert instances persisted by the alerts engine when the engine has been configured to use XML File persistence.
solutions/alerting/xml_persistence/ notifications	This directory contains notifier definitions persisted by the alerts engine when the engine has been configured to use XML File persistence. Each file represents a specific notifier that is able to send alert notifications to a user.
solutions/alerting/xml_persistence/ notifier_defs	This directory contains notifier definitions persisted by the alerts engine when the engine has been configured to use XML File persistence. Each file represents a specific notifier and aggregates all of the information the alerts engine needs to store about how to use that notifier to send alerts to users through the channel represented by the definition.
solutions/alerting/xml_persistence/ user_alerts	This directory contains user-customized alert instances persisted by the alerts engine when the engine has been configured to use XML File persistence.
solutions/alerting/xml_persistence/ user_contexts	This directory contains user contexts persisted by the alerts engine when the engine has been configured to use XML File persistence. Each file represents a specific user and aggregates all of the information the alerts engine needs to store about that user to serve their alerts.

Path	Description
work/lib	Contains various Java class libraries needed by the alerts engine at runtime. The library named ″alerts_engine.jar″ contains the alerts engine implementation itself. You will need to include this library in your project's build path when you write Java-based alert evaluators.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Office 4360
One Rogers Street
Cambridge, MA 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Supplemental Copyrights and Acknowledgments

The following Copyrights and Acknowledgments apply to various software components included in IBM WebSphere Portlet Factory, but neither created nor owned by IBM.

This offering is based on technology from the Eclipse Project (http://www.eclipse.org/).

This product includes software developed by the Apache Software Foundation (http://www.apache.org).

Portions of this software provided by JDOM. Copyright (C) 2001 Brett McLaughlin & Jason Hunter. All rights reserved. Portions of this software provided by the

membership of the XML-DEV mailing list. You can obtain a copy of the SAX copyright status at http:// www.megginson.com /SAX/.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM
Lotus
Portlet Factory
WebSphere

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Program Number: 5724-S21

Printed in USA

SC23-5922-00