Creating and implementing a notification plugin

Notifications can be set to alert you if a job is successful, unsuccessful, or either result. Additionally, a notification can also indicate if a job was launched from the Workbench or from a command line, and can produce a report in a number of different file formats.

You can select the following options and option parameters using the job notification plugin:

Notification Option	Option Parameters
Notify When Run From	• Workbench • Command Line • Both
Attached Report	• None • CSV • XML • HTML • PDF • RTF
Send Notification	• Always • Only on Success • Only on Failure

Notification parameters are set independently on each job configured in DB Change Manager. Additionally, each set of notification options apply individually per job selected.

The notification plugin must be defined in Eclipse and then activated in DB Change Manager. The following tasks provide a high-level overview of defining and implementing the notification plugin:

Create a plugin project

In order to start building a plugin, you need to create a new plugin platform in Eclipse.

To create a plugin project

1. Select **File > New > Other**.
   The Select a Wizard dialog appears.
2. Select **Plugin Project**, and then click **Next**.
3. Type a name for the plugin in the appropriate field, and leave the remainder of the parameters as they appear.
4. Click **Next**.
5. Retain the default parameters on the next dialog, and then click **Finish**.
   The new plugin project is created in Eclipse and is ready for plugin development.

Configure the target platform

The target platform for the plugin development process needs to be identified in DB Change Manager. This indicates to the new plugin for what product it extends, and grants the plugin access to the system.

To configure the target platform

1. Select **Window > Preferences**.
   The Preferences dialog appears.
2. Choose the **Plugin Development > Target Platform** node, and then click **Browse**.
3. Navigate to the install directory for DB Change Manager.
4. Click **Apply**, and then click **OK**.
   The Preferences dialog closes and the target platform of the plugin is now indicated.

Define dependencies

The new plugin requires a pair of dependency definitions on the DB Change Manager Notification plugin:

- `com.idera.change.notifications`
- `org.eclipse.ui.forms`

1. Navigate to the `META-INF` folder in your project and double-click `MANIFEST.MF`.
   The `MANIFEST.MF` file opens in Eclipse.
2. Select the Dependencies tab, and then click **Add**.
3. Choose `com.idera.change.notifications`, and then click **OK**.
   The dependency is added.
4. Choose `org.eclipse.ui.forms`, and then click **OK**.
   The dependency is added.
5. Press *Ctrl+S* to save the changes; or choose **File>Save** from the menu to retain the new dependencies, and then close the editor.

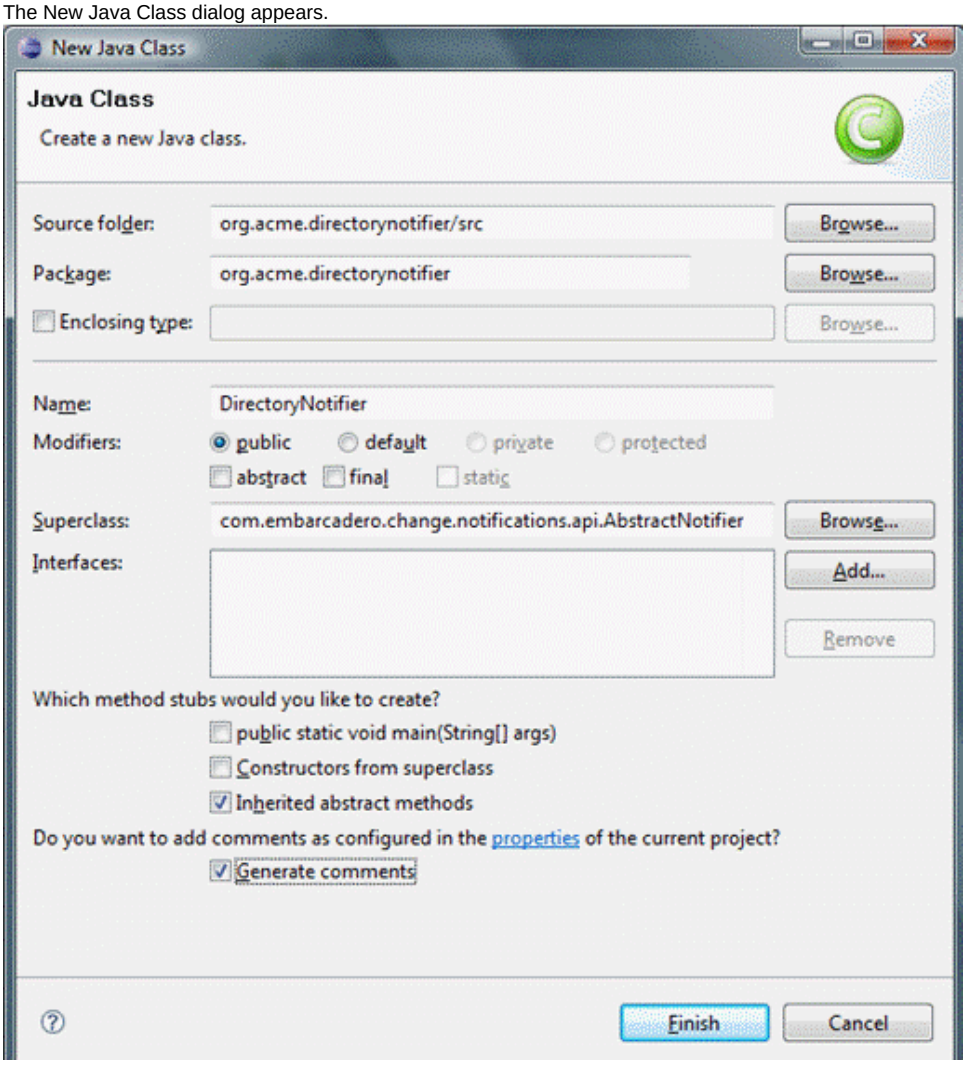
Implement a notifier class

Each notification plugin requires a `Notifier` class that must implement the interface `com.idera.change.notifications.api.INotifier`. To do this, you subclass `com.idera.change.notifications.api.AbstractNotifier` and customize it.

> If `AbstractNotifier` or `INotifier` cannot be found in Eclipse, the target platform or plugin dependencies are not configured properly. Ensure that you have configured these prerequisites and then search again for the notifier classes.

To implement a notifier class

1. Right-click on your project, and then select **New > Class**.
   The New Java Class dialog appears.



2. Enter the appropriate information in the fields provided to define the new Java class:

a. In the **Name** field, type "DirectoryNotifier".
b. In the **Package** field, type "org.acme.directorynotifier".
c. In the **Superclass** field, type "com.idera.change.notifications.api.AbstractNotifier".

3. Click **Finish** to create the new class.

Once you have defined the `DirectoryNotifier` class, two methods require implementation:

- `isReportSupported` returns a Boolean value indicating if the notification can include a report. The directory notifier returns true, because reports can be replaced in a directory, but a notification may not support reports. This method is called when creating job editors to determine if the reporting section is shown for this type of notification.
- `sendNotification` is called after a job runs and is responsible for the notification. The three parameters are as follows:

Parameter	Type	Description
`notifierData`	iNotifierData	This instance contains the configuration for the notification of the job just run.
`jobMetaData`	Map<String, String>	This map contains information about the job execution. The keys in the map correspond to entries in the NotificationPropertyEnum. For example, to get the data and time of the execution, you would code: "jobMetaData.get(NotificationPropertyEnum.DATE_TIME.getTag())" NOTE: The keys are the tag of the enum entries, so ensure that getTag() is called when accessing jobMetaData.
`notificationInfo`	iReportGenerator	Used to generate the report and access any DDL or sync script output.

Implement a viewer contribution

Each notifier can be configured for a job in DB Change Manager.

The notifier enables a user to select a target directory in which to put results. The email notifier that is supplied with the application enables users to select the email template that will be used to generate the message as well as the addresses of those who will receive it.

In order to add a directory field to the job editor, you write a class that implements the interface named `com.idera.change.notifications.api.INotifierViewerContribution`.

- Create a new class named "DirectoryNotifierViewerContribution" and follow the same steps you used to create the `DirectoryNotifier` class. The exception to this process is that you need to leave the Superclass parameter set to "java.lang.Object", as well as adding the interface `INotifierViewerContribution` to the list of implemented interfaces.

Specify setup constants

The notifier only allows users to select a directory in which reports and outcomes are written whenever jobs are executed.

The `DirectoryNotifierConstants` class is a small constants class that stores the property name for storage and retrieval purposes.

1. Create a new class named `DirectoryNotifierConstants`.
2. Enter the following code for the class definition:

```
/**
          * Constantsforthisnetworksharenotifier
          */
          publicfinalclassDirectoryNotifierConstants
          {
                    //Overviewfilename publicstatisfinalString OVERVIEW_FILE_NAME="outcome.txt";
                   //PersistedJobSettingsKey publicstatisfinalSTringTARGET_DIRECTORY="directorynotifier.
targetdir";
          /**
          * Constantutilityclasscannotbeinstantiated
          */
          privateDirectoryNotifierConstants()
          {
          }
}
```

Implement the DirectoryNotifier class

The `DirectoryNotifier` class can be coded with two return methods:

- `isReportSupported`
- `SendNotification`

To implement the `isReportSupported` method

- Enter the following code:

```
public boolean isReportSupported()
{
     return true;
}
```

To implement the `SendNotification` method

1. To add the `notifierProperties` and assign the `targetDirectory` to a variable to provide a directory that the user selected for the executed job, add the following code:

```
     public void sendNotification (INotifierData notifierData, Map<String, String> jobMetaData,
notificationInfo
     Map<String, String> notifierProperties = notifierData.getNotificationProperties();
     String targetDirectory = notifierProperties.get (DirectoryNotifierConstants.TARGET_DIRECTORY);
```

2. Enter the following code to define that the report and output file need to be placed in a directory whose name is composed of the job execution date and time.

> If a directory cannot be created, the code below also includes a `RuntimeException`.

```
//Create a new directory named by the execution date and time
     String dateTime = jobMetaData.get (NotificationPropertyEnum.DATE_TIME.getTag() );
     File dir = new File (targetDirectory +  File.seperatorChar + dateTime.replace ( ':', '-' ) );
     boolean createdDirectory = dir.mkdir();
     if (!createdDirectory )
     {
     throw new RuntimeException ("Create Failed: " +  dir.getAbsolutePath () );

}
```

3. Enter the following code to define the outcome file, which includes each of the job execution parameters:

```
try
{
     //Write the results to an output file
     File f = new File (dir.getAbsolutePath() + File.seperatorChar +
     DirectoryNotifierConstants.OVERVIEW_FILE_NAME );
     f.createNewFIle();

     BufferedWriter writer = new BufferedWriter (new FileWriter (f) );
     String newLine = System.getProperty ("line.seperator");
     writer.write ("Outcome: " + jobMetaDAta.get
          (NotificationPropertyEnum.JOB_OUTCOME.getTag() ) + newLine);
          writer.write (newLine);
     writer.write ("Date & TimeL" + jobMetaData.get (
          NotificationPropertyEnum.DATE_TIME.getTag() ) + newLine);
     writer.write ("Elapsed Time: " +  jobMetaData.get (
          NotificationPropertyEnum.ELAPSED_TIME.getTag() ) + newLine);
          writer.write newLine );
     writer.write ("Name: " + jobMetaDAta.get (
          NotificationPropertyEnum.JOB_NAME.getTag() ) + newLine);
          writer.write (newLine);
     writer.write ("Host: " + jobMetaData.get (
          NotificationPropertyEnum.JOB_HOST.getTag() ) + newLine);
     writer.write ("Job Type: " + jobMetaData.get (
          NotificationPropertyEnum.JOB_TYPE.getTag() ) _  newLine);
     writer.write ("Module: " + jobMetaData.get (
          NotificationPropertyEnum.MODULE.getTag() ) +  newLine);
          writer.write (newLine);
     writer.write ("Sources: " + jobMetaData.get (
          NotificationPropertyEnum.JOB_SOURCES.getTag() ) + newLine);
     writer.write ("Targets: " + jobMetaData.get (
          NotificationPropertyEnum.JOB_TARGETS.getTag() ) + newLine);
          writer.write (newLine);
     writer.write ("Job Notes: " + jobMetaDAta.get (
          NotificationPropertyEnum.JOB_NOTES.getTag() ) + newLine );
          writer.flush();
          writer.close();
}
catch (Throwable t)
{
     throw new RuntimeException (t);
}
```

4. Enter the following code to generate the report. This code also provides a null report if the report type is set to none at execution time.
```
//Create the report if required
File reportFile = notificationInfo.getReport() ;

if (reportFile !=null)
{
     File destinationReportFile = new File dir.getAbsolutePath() + File.seperatorChar +
     reportFile.getName()); eportFile.renameTo (destinationReportFile);
}
```
5. Finally, output any DDL or synchronization SQL:
```
//Write the script File scriptFile=notificationInfo.getScript();

if ( scriptFile !=null) File DestinationScriptFIle = new File (dir.getAbsolutePath() +
     File.separatorChar + scriptFile.getName () );
     scriptFile.renameTo (destinationScriptFile);
```

Implement the `DirectoryNotifierContribution` class

Add the code below to the `DirectoryNotifierViewerContribution` class to provide functionality for the interface:

1. The `DirectoryNotifierViewerContribution` class requires three class-level private attributes. These attributes store notification data, which is the composite on which any options are placed, as well as the `folderField`, which is used to enter a file directory address.
```
private INotifierData notificationData;
private Composite body;
private Text folderField;
```
2. The `createFormContent` attribute creates a new composite and provides a layout template for options and entry fields. The code below adds a label and a text field to the interface:
```
public Composite createFormContent (Composite parent, FormToolkit toolkit) body = toolkit.
createComposite (parent);
body.setLayout (new GridLayout ());

     toolkit.createLabel (parent, "Target Directory:");

     String folder = notificationData == null ? " " " : notificationData.getNotificationProperties().
get (DirectoryNotifierConstants.TARGET_DIRECTORY);

     folderField = toolkit.createText (parent, folder, SWT.SINGLE | SWT.BORDER);
     folderField.setLayoutData (new GridData (SWT.FILL, SWT.CENTER, true, false));
     folderField.addModifyListener (new ModifyListener () {
          public void modifyText (ModifyEvent e)
     {
          notificationData.setNotificationPropert (DirectoryNotifierConstants.TARGET_DIRECTORY,
folderField.getText().trim() );
     }
});
return body;
```
The code does the following:
   - Creates a new composite body, and sets it to have a grid layout.
   - Creates a label for the target directory field.
   - Retrieves the folder from the job, if the job is new or unsaved, or defaults to an empty string if the result is null.
   - Initializes the `folderField` text element and sets it to justify.
   - Adds a listener for changes on the `folderField` element that updates the `notificationData`, thus enabling save actions.
   - Returns the body to be added to the DB Change Manager job editor to which this class applies.
3. Add the `setData` method using the following code, which is called to update the user interface with job information. This method accepts and stores the new instance of `INotifierData` containing job-specific data for the notifier. If the `folderField` has already had something entered, the notification properties are initialized and store the value that was previously entered.
```
public void setData (INotifierData notificationData)

     this.notificationData = notificationData;

     if (folderField !=null && !folderField.isDisposed() && folderField.getText().trim().length() > 0))
          if (notificationData !=null) Map<String, String> propMap notificationData.
getNotificationProperties();
          if (propMap == null)
          {
               notificationData.setNotificationProperties(new HashMap<String, String>());

               propMap = notificationData.getNotificationProperties();
          }
          notificationData.setNotificationProperty (DirectoryNotifierConstants.TARGET_DIRECTORY,
folderField.getText());
          }
     }
}
```

Register the extensions

Once you have coded the plugin and implemented the notifier and user interface, you include definitions that explain how DB Change Manager should interact with the plugin.

You register the extensions that define the `Notifier` and `NotifierViewerContribution` to DB Change Manager. When the plugin is added to DB Change Manager, the extensions are registered and DB Change Manager understands the purpose of the plugin.

1. Open the `MANIFEST.MF` file.
2. Navigate to the plugin.xml tab and display the source.
3. Add the following code:
```
<extension point="com.idera.change.notifications.notifier">
     notifier
          id="org.acme.directorynotifier" name="Directory Notifier"
          notifier="org.acme.directorynotifier.DirectoryNotifier"
          viewContribution="org.acme.directorynotifier.DirectoryNotifierViewerContribution">
     </notifier>
</extension>
```
4. Save the file by pressing *Ctrl+S* or by clicking **File > Save**.

When `org.acme.directorynotifier` is found, the notification extension point is registered. The `notifier` and `viewContribution` attributes are the class names for two implementations and Eclipse reflectively registers these objects for use in DB Change Manager.

Deploy the `Notifier` plugin

Once the plugin has been defined, you need to deploy the plugin in DB Change Manager.

1. Stop DB Change Manager if it is currently running.
2. Select **File > Export**.
   The Export dialog appears.
3. Choose **Deployable Plugins and Fragments** from the **Plugin Development** group.
4. Click **Next**.
5. Select `org.acme.directorynotifier` and ensure that the target directory is attributing to the DB Change Manager installation folder.
6. Click **Finish**.
7. Start DB Change Manager, and then create a new data comparison job.
8. Define a source and target data source, and then navigate to the Notifications tab.
   The Directory Notifier appears as a section to the tab.
9. Choose **Enable Directory Notifier** to enable the feature.
10. Specify the **Target Directory**, such as **C:\ChangeManagerNotifications**, and then choose **PDF** in the **Attached Report** section.
11. Execute the job, and then open the target folder. A new folder is created containing two output files from the job.