Creating and implementing a custom configuration plugin

Follow the instructions in these sections to code an Eclipse plugin for DB Change Manager that adds additional configuration parameters and values for configuration standards, archive jobs, and comparisons.

Create a project plugin

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. Leave the parameters on the next screen as they appear, 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. The target platform indicates to the new plugin for what product it extends, and grants the plugin access to the system.

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, and then click **Apply**.
4. Click **OK**.
   The Preferences dialog closes and the target platform of the plugin is now indicated.

Define dependencies

The plugin requires a dependency for configuration comparison jobs in DB Change Manager. This dependency can be set by opening the `MANIFEST.MF` file located in the `META-INF` directory of your project.

1. Select the Dependencies tab, and then click **Add**.
2. Choose `com.idera.change.config.propertysource`, and then click **OK**.
   The dependency is added to the plugin and appears in the **Dependencies** list.
3. Press *CTRL-S* or click **File>Save**.
   Once the dependency definition is saved, you can close the editor and the system retains your selection.

Implement a property source

Each configuration property source plugin must contain a class that extends com.idera.change.config.propertysource.api.ConfigurationPropertySource.

If ConfigurationPropertySource cannot be found in Eclipse, it means you have misconfigured the target platform or target dependencies.

1. Right-click the project, and then select **New>Class**.
   The New Java Class dialog appears.
2. Name the class `JDBCVersionInformation`, set the package to `org.acme.versioninformation`, and then set the superclass to `com.idera.change.config.propertysource.api.ConfigurationPropertySource`.
3. Click **Finish**.
   The new class is generated.

Determine support

Each class that subclasses `ConfigurationPropertySource` contributes properties to the configuration comparison job functionality.

The implementation uses the `isSupported` method, which determines if there are attributes available for a specific data source.

For example, if a user added the Sybase @@Version tag as a configuration property, it would be pointless to run the code for Oracle, SQL Server, or DB2 for LUW data sources.

The method has the following signature:

```
boolean isSupported (IDatasourceIdentifier dataSourceIdentifier)
```

This method is called by DB Change Manager for any registered content plugin when a data source is used in a configuration archive job, a standard, or a comparison job. When it encounters a data source identifier, it returns true or false and indicates whether the property source provides parameters for the data source.

Implement the following code to define this method:

```
public boolean isSupported (IDatasourceIdentifier dataSourceIdentifier)
{
     return dataSourceIdentifier.getDatabaseVersion().getDBMSType() !=DBMSType.DB2;
}
```

Define properties

The `getAvailableProperties` function returns a set of instances of `IConfigurationProperty` instances.

There are several utility methods to help instantiate these instances, named of the form, `createConfigurationProperty`.

Implement the following code to define the `getAvailableProperties` method:

```
public Set(IConfigurationProperty> getAvailableProperties (IDatasourceIdentifier identifier)
{
     Set<IConfigurationProperty> results = new HashSet<IConfigurationProperty>();
     for (EVersionProperties prop : EVersionProperties.values())
     {
          results.add (createCOnfigurationProperty (prop.getDisplayName(), prop.name(),"", prop.getType(),
"" ));
     }
return results;
}
```

Because the `EVersionProperties` code cannot be resolved, you can add a Java enum to define the properties provided by the plugin.

Create a new file named `EVersionProperties.java` and implement the following code:

```
package org.acme.versioninformation;
import java.sql.Connection;
import java.sql.SQLException;
import com.idera.change.config.propertysource.api.ConfigurationPropertyType;
/**
* An enumeration of version properties
*/
public enum EVersionProperties
{
     PRODUCT_NAME (ConfigurationPropertyType.STRING, "Product Name")
     {
          /**
          * @returns the JDBC metadata's information for product name
          * @throws SQLException if ther eis a JDBC error
          */
          public String getValue (Connection connection) throws SQLException
          {
               return connection.getMetaData().getDatabaseProductName();
          }
     },
     PRODUCT_VERSION (ConfigurationPropertyType.STRING, "Product Version")
     {
          /**
          * @returns the JDBC metadata's information for product version
          * @throws SQLException if there is a JDBC error
          */
          public String getVAlue (Connection connection) throws SQLException
          {
               return connection.getMetaData().getDAtabaseProductVersion();
          }
     },
     MAJOR_VERSION (ConfigurationPropertyType.NUMERIC, "Major Version")
```

```
     {
          /**
          * @returns the JDBC metadata's information for major version
          * @throws SQLException if there is a JDBC error
          */
          public String getValue (Connection connection) throws SQLException
          {
               return "" + connection.getMetaData().getDataMajorVersion();
          }
     };
//Attributes private ConfigurationPropertyType type; private String name;
/**
* Constructor to set entry attributes
* @param type the type
* @param name
*/
private EVersionProperties (ConfigurationPropertyType type, String name)
}
{
/**
     this.type = type;
     this.name = name;
* Given a JDBC connection return the parameter's value
* @param connection the database connection
* @throws SQLException if there is a JDBC error
*/
abstract String getValue (Connection connection) throws SQLException;
/**
* @return the display name of the parameter
*/
public String getDisplayName()
{
return name;
}
/**
* @return the type of the parameter
*/
public ConfigurationPropertyType getType()
{
return type;
}
/**
* Finds a property of a name and returns it
* @param name the name of the property
* @return the property
*/
public static EVersionProperties getProperty (String name)
{
     for (EVersionProperties p : values())
     {
          if (p.name().equals (name))
          {
               return p;
          }
     }
     throw new RuntimeException (EVersionProperties.class.getName() + "not found for '" + name + "'");
     }
}
```

In the code above, the four entries that capture from JDBC the product name, version, minor version, and major version are added to the enum.

Gather results

DB Change Manager gathers results when it collects all of the properties and values for display purposes or to store in an archive. The extraction method is called when a data source is indicated in a standard, or when an archive or comparison job is run.

```
public Set<IConfigurationPropertyResult> extract(Set<IConfigurationProperty> properties,
IDatasourceIdentifier dataSourceIdentifier)
```

A set of `IConfigurationPropertyResult` instances that correspond to the properties must be returned, and values need to be added for the properties returned from the data source, as follows:

```
public Set<IConfigurationPropertyResult> extract (Set<IConfigurationProperty> properties,
IDatasourceIdentifier dataSourceIdentifier)
{
     try
     {
          Set<IConfigurationPropertyResult> results = new HashSet<IConfigurationPropertyResult> ();
          Connection connection = getConnection (dataSourceIdentifier);
          for (IConfigurationProperty configProp : properties)
          {
               EVersionProperties prop = EVersionProperties.getProperty (configProp.getPropertyIdentifier());
               String value = prop.getValue (connection);
               results.add (createConfigurationPropertyResult (configProp, true, value)
               );
          }
     return results;
     }
catch (SQLException sql)
{
     throw new RuntimeException (sql);
     }
}
```

In the above implementation, the code creates an empty set to hold the results and then retrieves the JDBC connection to the data source. For each retrieval, the enum constant `EVersionProperties` on which `getValue` is called to retrieve each property value from JDBC metadata. Finally, `createConfigurationPropertyResult` is called to create the result instance.

Synchronize the job

The synchronization method returns an empty array because users cannot synchronize version information.

```
public List<IPropertyResultsSynchronization> sync (IDatasourceIdentifier dataSourceIdentifier,
     List<IConfigurationPropertyResult> sourcePropertyValues, List<IConfigurationPropertyResult>
     targetPropertyValues)
{
     return new ArrayList<IPropertyResultsSynchronization>();
}
```

Register extensions

Once you have completed coding the plugin, register it with DB Change Manager by implementing an Eclipse extension. When the plugin is added to DB Change Manager, the extensions are registered and DB Change Manager understands that the plugin adds a new type of notification to the module.

To register extensions

1. Open the `Manifest.MF` file and navigate to the Extensions tab.
2. Click **Add** to register an extension.
   The New Extension dialog appears.
3. Choose `com.idera.change.config.propertysource.contribution`.
4. Click **Finish**, and then close the dialog.
   The extension is added to the **All Extensions** list.
5. Select the (source) entry on the right-hand side of the screen, and then enter the following details:
   - **display-name**: JDBC Metadata version information
   - **source-identifier**: `org.acme.jdbc.versioninformation.source`
   - **class**: `org.acme.jdbcversioninformation.JDBCVersionInformation`

   > The class name is used to enable DB Change Manager to instantiate the new class, while the display-name is the group under which the four JDBC metadata version properties appear.

6. Save the file by pressing *CTRL-S* or by choosing **File>Save** from the menu.
   The class notification is complete, and DB Change Manager and Eclipse are informed about the property source. When the plugin is added to DB Change Manager, it will add your properties to configuration comparison and standard jobs.

Deploy the custom configuration plugin

To deploy the custom plugin

1. Shut down DB Change Manager if it is running.
2. Click **File>Export**.
   The Export dialog appears.
3. Select **Deployable Plugins and Fragments** from the **Plugin Development** group, and click then **Next**.
4. Select `org.acme.jdbcversioninformation` and ensure the target directory is the same as the DB Change Manager installation folder.
5. Click **Finish**.
   The new plugin is deployed.

6. Start DB Change Manager and create a new configuration archive job. When you select an Oracle, Sybase, or SQL Server data source, the Refinements and the Results tabs contain the four version properties you defined in the new plugin.