Oracle® Enterprise Data Quality for Product Data

Java API Interface Guide

Release 11g R1 (11.1.1.6)

E29133-03

February 2013

ORACLE®

Oracle Enterprise Data Quality for Product Data Java API Interface Guide, Release 11g R1 (11.1.1.6)

E29133-03

Contents

Preface

This guide is intended to explain the basic capabilities of the Oracle DataLens Server Java Interface.

To understand all of the features presented, you must use this guide in conjunction with the Oracle Enterprise Data Quality for Product Data documents listed in "Related Documents" on page 2-vii.

Audience

You should have a basic understanding of the DataLens Technology. You must be an experienced Java software developer.

This document is intended for all users of the DataLens Technology, including:

- Subject Matter Experts (SMEs)
- IT Administrators

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For more information, see the following documents in the documentation set:

- The Oracle Java API documentation (Javadoc) provides a list of all packages, including those used in the Oracle Enterprise Data Quality for Product Data Java API, with a summary for each. Javadoc is delivered in the Oracle Enterprise Data Quality for Product Data Developer Tool Kit package in the `...\DevToolKit\java_api\doc\javadoc` directory. You can access Javadoc by locating this directory and opening the `index.html` file in a browser.

- The *Oracle Enterprise Data Quality for Product Data Oracle DataLens Server Installation Guide* provides detailed Oracle DataLens Server installation instructions.
- The *Oracle Enterprise Data Quality for Product Data Oracle DataLens Server Administration Guide* provides information about installing and managing an Oracle DataLens Server.
- The *Oracle Enterprise Data Quality for Product Data Application Studio Reference Guide* provides information about creating and maintaining Data Service Applications (DSAs).

See the latest version of this and all documents in the Oracle Enterprise Data Quality for Product Data Documentation Web site at

http://docs.oracle.com/cd/E35636_01/index.htm

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, text that you enter, or a file, directory, or path name.
`monospace`	Boldface, monospace type indicates commands or text that you enter.

1

Overview Oracle DataLens Server APIs

This chapter provides information about the Enterprise DQ for Product (EDQP) Oracle DataLens Server Java application programming interface (API) and includes the following:

Application Programming Interfaces

There are three main APIs to the Oracle DataLens Server platform.

- **DSA Client** - The Oracle DataLens Server DSA Client interface. This is used for direct access to the DSA loaded on the Oracle DataLens Servers for processing application/enterprise data. The DSA Client can process the following:
  - Tab-separated input data
  - Input data with any user-defined separator character
  - Input data from a database query
- **Information Client** - The Oracle DataLens Server Information Client interface. This is used for access to information about the data lenses, Transform Maps, and DSAs that are loaded on the Oracle DataLens Servers.
- **Ping Client** - The Server Availability Client interface. This is used to check for a response from any Oracle DataLens Server in an Oracle DataLens Server Group.

There is also a low-level Application Programming Interface to the Oracle DataLens Server platform. This interface is not recommended for application developers.

Platforms

The Java API can be used for integrating to the following.

- Java applications or Web pages
- Available on Windows or Linux operating systems

Pre-Installation Requirements

Java JDK 1.6.0_29 - The API was compiled and built using this release.

Java JDK 1.5, 1.6 - The API is compatible with these releases.

Oracle DataLens Server Java Libraries

The libraries and software used by this API are delivered in the product download in the `opdq-devtoolkit.zip` file. The following libraries are provided, in the `\DevToolKit\java_api\lib` directory, for creating new applications and integrating to existing applications:

`opdq-api1.jar`
This library contains the Application interface classes.

`jdom-1.0.jar`
This library contains the third party component, Jdom 1.0, needed by the API. Jdom is used for encoding/decoding the SOAP messages sent to the Oracle DataLens Server from the client application.

2

Using the DSA API with the Oracle DataLens Server

This chapter describes how the DSA API is used with the Oracle DataLens Server.

WfgClient

Use of this interface is the preferred method to access any data transformations that need to be done on the Oracle DataLens Server. This interface will directly execute the DSAs that are deployed to the Oracle DataLens Server. Use of this interface should supplant use of the Oracle DataLens API.

The `WfgClient` class is used as an interface to the Oracle DataLens Server. This class provides methods to perform DSA transformations.

Updating Individual Records and Data Lines

DSAs are not a one-to-one match of the input records and the output records. In some cases this may be true, depending on the map. More likely, there will be multiple output steps, and each step will only have a subset of the input data results. In some output steps, there may be no data returned, and in other cases there may be multiple output records returned for a single input record.

This means that the DSAs should pass the original Id into the processing, usually as the first data field. This provides a means for matching the output result data with the original input data.

In cases where data is just being processed, and there is no need to link the results back to each individual input record, then passing the ID through the DSA is not needed.

Transforming Data

Use the following to transform data:

Import

Import the `WfgClient` with the following lines:

```
import oracle.pdq.api1.api.client.WfgClient;
import oracle.pdq.api1.api.client.WfgResultLine;
import oracle.pdq.api1.api.client.WfgRequestLine;
import oracle.pdq.api1.api.bean.Fault;
import oracle.pdq.api1.api.iface.ErrorIF;
import oracle.pdq.api1.api.util.Priorities;
```

Initialize the Client

An instance of the WfgClient class needs to be created with the Oracle DataLens Server name and port.

Actual parameters:

- **Server Name** - This can be either a machine name (such as, "Production") or an IP address (such as "127.0.0.1").
- **Server Port** - This is the port number of the server. By default, the Oracle DataLens Server is installed on port 2229.
- **Encryption flag** - False uses normal HTTP communication; true uses the secure HTTPS.
- **Client Code** - This is the "secret code" that the Oracle DataLens Server provides with your server license to prevent unauthorized access to the Oracle DataLens Server via this API. This code is built into the Oracle DataLens Server license and is only active if requested as part of the license. This value can be left blank if the server license has no code.
- **Application** - This application name initiated the client request to the server. This name is used to accumulate server statistics on the Oracle DataLens Server Administration Web Pages.

```
// Create WfgClient object
WfgClient wfgClient;
        wfgClient = new WfgClient(serverName, SERVER_PORT, ENCRYPTION,
                                                clientCode, APPLICATION);
```

Create the List of Input Data

This is a brief example of creating the input data list. First, you create the list from an array of static data as shown as follows:

```
private static final String m_inputData[][] = {
{"0", "Res, 20 Ohm"},
        {"1", "Res, Net 4 W"};
```

This data above is just an example. Your data will come from your application, from an input file or a database query.

In any case, the data needs to be put into an input list for the Java API to process the data. Following is an example of creating and populating the list using the example data. In this case, the input data needs to be separated using the character separator, in this case the Tab character. This interface is best used when there is only one field of input data to be processed.

```
// Setup this list of String Fields for the request
List list = new ArrayList();
    for (int i=0; i<m_inputData.length; i++) {
        List fields = new ArrayList(); // Create a List of Strings
        fields.add(new String(m_inputData[i][0])); // Add the ID Data Field
        fields.add(new String(m_inputData[i][1])); // Add the Description Field
        list.add(fields);
    }
```

Transform a List of Data

A list is passed to the `runRtJob` method and a single job ID is returned. The `runRtJob` method is called just a single time with a single list of data.

Actual parameters:

- **Job ID** - The DSA Job ID obtained from the `runJob` call.
- **DSA Name** - The name of the DSA to run on the Oracle DataLens Server.
- **Description** - A description of this particular job.
- **List** - The list with a list of String input fields.

```
// Start the DSA job with our data
// NOTE: Input data with a List containing a list of string attributes.
    //       This is useful for already separated data
    m_wfgClient.setLinesFromFields(list);
    int m_jobID = m_wfgClient.runJob(PMapName, "My Job");
```

The preceding call is using the following default values:

- Job Priority of medium
- Job run-time locale of USA English

Alternative Method of Transforming Data

In any case, the data needs to be put into an input list for the Java API to process the data. Following is an example of creating and populating the list using the example data. In this case, the input data needs to be separated using the character separator. The following example uses the Tab character. This interface is best used when there is only one field of input data to be processed.

```
List list = new ArrayList();
for (int i=0; i<m_inputData.length; i++) {
    list.add(new WfgRequestLine(m_inputData[i][0] + "\t" + m_inputData[i][1]));
}
```

Now process the data using the list you created:

List

The list of `WfgRequestLine` objects that have been initialized with the tab-separated input data.

```
// Start the DSA job with our data
wfgClient.setLines(list);
       int m_jobID = wfgClient.runJob(PMapName, "Comment: API Test Job 1");
```

Retrieve Results from the Server for Jobs with a Single Output Step

When the job has finished, the transformed data can be retrieved from the server back to the client application. The `getResultData` method is called just a single time and returns a list of `WfgResultLine` objects containing the result data.

Synchronous Method

The call will wait until the job has finished processing the data before control is returned to the program with the result data.

```
// Get the DSA Results!
```

```
boolean waitForResults = true;
resultData = wfgClient.getResultData(jobID, waitForResults);
```

Asynchronous Method

You can check the job status and do other processing while waiting for the job to complete. The `getResultData` method will throw a fault indicating that the job is still processing the input data.

```
try {
// Get the DSA Results!
        boolean waitForResults = true;
        resultData = wfgClient.getResultData(jobID, waitForResults);
    } catch (Fault f) {
        // Check if the job has not completed yet
       if (f.getErrorCode() == ErrorIF.ERROR_NOT_COMPLETED)
          . . .
  }
```

Server Faults that can be thrown from a call to `getResultData` include the following `ErrorIF` errors:

```
ERROR_JOB_CANCELED

ERROR_CANCEL_FAILED

ERROR_COPY_FAILED

ERROR_JOB_FAILED

ERROR_NOT_COMPLETED
```

Retrieve Results from the Server for Jobs with Multiple Output Steps

When the job has finished, the transformed data can be retrieved from the Server back to the client application. The `getResultData` method is called just a single time for each output step. Each call returns a list of `WfgResultLine` objects containing the result data, just as with the jobs with a single output step.

The call will wait until the job has finished processing the data before control is returned to the program with the result data.

```
// Get the DSA Results!
resultData = wfgClient.getResultData(jobID, stepName, waitForResults);
```

The call to `getResultData` can be made synchronously or asynchronously as demonstrated above.

Pulling the Result Data from the List

The result data is returned as a list of `WfgResultLine` objects.

List Data

This is how the result data fields should be pulled from the output lines. This list interface will always maintain all the columns of output data, even if there is no data for a particular output data field. In this case, the data field result will be a null value.

The following code excerpt demonstrates pulling out the individual data lines, with the individual data fields.

```
// Iterate through the result data lines
Iterator iter = resultData.iterator();
    while (iter.hasNext()) {
        WfgResultLine resultLine = (WfgResultLine)iter.next();
        List outFields = resultLine.getDataFields();

        // Iterate through the result data fields
        Iterator i2 = outFields.iterator();
        while (i2.hasNext()) {
            String outField = (String)i2.next();
            System.out.print(outField);
            If (i2.hasNext())System.out.print(", ");
        }
         System.out.println(" ");
    }
```

Tab-Separated Data

This is a simple way to get to the result data for testing. The following code excerpt demonstrates pulling out each line of tab-separated output data.

> **Note:** This example works if you have specified an alternate separator character.

```
Iterator iter = resultData.iterator();
while(iter.hasNext()) {
            WfgResultLine resultLine = (WfgResultLine)iter.next();
            System.out.println(resultLine.getData());
        }
```

Listing Multiple DSA Jobs

The DSA Client can list Jobs can be listed from the Oracle DataLens Server Administration Web Pages. The following types of lists can be retrieved from the server.

- All Jobs (also since a particular date)
- All jobs that have not completed
- All jobs for a particular submitter (also since a particular date)
- All not-completed jobs for a particular submitter
- All jobs for a particular approver

The following code shows the calls in the order listed in the preceding:

```
List list = wfgClient.listAllJobs(sinceTS);
List list = wfgClient.listNCJobs();
List list = wfgClient.listSubmitterJobs(submitter, sinceTS);
List list = wfgClient.listNCSubmitterJobs(submitter);
List list = wfgClient.listApproverJobs(approver);
```

These calls all return lists of `WfgJobInfo` objects.

Listing a Single DSA Job

Information can also be obtained from a single job given the Job ID. The following Java code example shows this:

```
WfgJobInfo jobInfo = wfgClient.listJob(jobID);
```

This call returns a single WfgJobInfo object with the job details.

Additionally, all the details on the steps are returned as well. To get the steps, use the getSteps method call as shown in the following example:

```
List steps = jobInfo.getSteps();
```

These steps are a list of WfgJobStepInfo objects with all the details on the individual job steps.

Using File Input and Output

The DSA API can use a text file as input and a text file as output. The complete path to the input file and the complete path to the output directory are needed. Use the setters to toggle on the input/output directory locations as in the following example:

```
// Setting the input file and output directory toggles on file processing
wfgClient.setOutputDirectory(outputLocation);
wfgClient.setInputFilePath(filePath);
jobID = wfgClient.runJob(transformProcess, desc);
```

These file input paths and the file output paths are sent directly to the Oracle DataLens Server. This means that the paths must be paths that are relative to the server. For example, if you give the path to an input file as:

```
C:/temp/raw_data.txt
```

This file is from the C drive on the server machine, not the C drive on the client machine. The output directory is also a relative path from the server machine as well.

The source path can be a UNC path to a file on a remote machine.

Here is an example:

```
//node_name/shared/test.txt
```

Miscellaneous Settings for the WfgClient

These are options that can be used by the WfgClient. In fact, these settings can be used by any of the Oracle DataLens Server Client API classes. For a complete list of methods in the WfgClient class and additional information, see the Javadoc documentation as described in "Related Documents" on page 2-vii.

Retry Count

This is useful to control the amount of time that the client attempts to connect to the Oracle DataLens Server. The default is to retry 20 times. This could be a problem in an interactive user environment, where you does not have a couple of minutes while WfgClient is attempting to connect to the server. In these cases you could set the retry count to 1 or even 0. Look also at PingClient, which can be used to check if a particular server is responding.

```
// Just set the retry to one for starting the job, then use the default
```

```
wfgClient.setRetryCount(1);jobID = wfgClient.runRtJob(transformProcess,
jobPriority, desc, rtLocale, input);
wfgClient.setRetryCountToDefault();
```

Filter Data

By default, data filtering is turned on for all input data. This will filter out all inadvertent control characters that may be interspersed in your input data. This data can cause problems with processing and sometimes it can cause problems with sending the data from the client to the server via HTTP as XML Soap documents. Tab characters are never filtered out.

```
// By default filtering is turned on and nothing needs to be done
wfgClient.setFilterData(false);
jobID = wfgClient.runRtJob(transformProcess, jobPriority, desc, rtLocale, input);
```

In the preceding example, the parameter input (with the List of input data) will be filtered.

Where the filtering encounters control characters in the input data, they will be substituted with the "?" character. This facilitates you in tracking down the source and exact location of the control characters. The data lens can ignore the "?" character when processing the input lines.

Job Priority

By default, a job priority of medium is used for all jobs.

This is the priority the job will be given on the server for processing. Large batch overnight jobs should be given a priority of low. Small jobs with few input records, or requests that need a quick response, such as users waiting for a response should get a priority of High. All other jobs should use a priority of medium. The number of concurrent jobs that can be run on the server is also controlled by the priority of the job (For more information, use the Configuration link on the Oracle DataLens Server Administration Web Pages). These priority values can be used from the `Priorities` class in the `edqp-api.jar`.

- `Priorities.PRIORITY_LOW`
- `Priorities.PRIORITY_MEDIUM`
- `Priorities.PRIORITY_HIGH`

```
// Set the job priority
wfgClient.setPriority(Priorities.PRIORITY_HIGH);
```

Run-Time Locale

By default, a run-time locale of USA English (`en_US`) is used.

Set the locale to use for output of this job.

```
// Set the run-time locale
wfgClient.setRuntimeLocale(RT_LOCALE);
```

Separator Character

By default, a field separator character of tab is used.

```
// Set the run-time locale
wfgClient.setFieldSeparator('|');
```

> **Note:** If you are using a different separator character than the default, then the separator character must be specified when pulling the data fields from the WfgResultLine data object.

```
List fields = wfgResultLine.getDataFields(FIELD_SEPARATOR_CHAR);
```

Client-Side Debugging Toggle

By default, this is toggled off when a new WfgClient object is created

This will dump the client information out to standard output prior to sending the request to the server. This is only used for debugging and should never be toggled to on in a production environment.

```
// Toggle on client data to standard output
wfgClient.setTrace(true);
```

email Output

If set, then an API request that would return a list or update a file, will email the results to the user specified instead.

```
// Send the results to the following user
wfgClient.setEmailAddress("user1@systems.com");
```

A DSA that updates a database will continue to update the database.

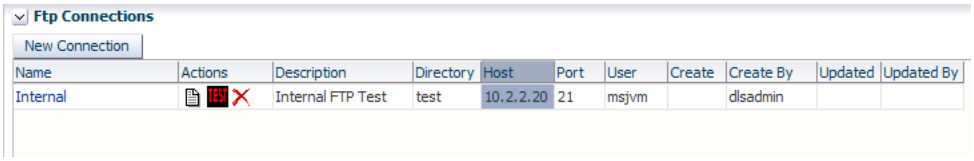
A DSA can be defined to return the results to an email address. This will work regardless of this API email setting. In fact, the email address in the DSA will take precedence over this email set in the API.

FTP Output

If set, then an API request that would return a list or update a file, will send the results instead to the FTP location specified.

```
// Send the results to the following FTP site
wfgClient.setFtpName("internal");
```

This value should not be set if setEmailAddress is being used. In addition, the FTP name, internal, must be setup on the Oracle DataLens Server as in the following:



Database Parameters

By default, database parameters are not used.

This is used where the input map is expecting input from a database query and the query requires parameters that must be passed in.

Create a list of parameters and then set the database parameters as shown in the following code excerpt:

```
// Set the database parameters
List dbParams = new ArrayList();
            dbParams.add("first_parameter");
            dbParams.add("second_parameter");
            wfgClient.setDbParameters(dbParams);
```

3

Server Information API to the Oracle DataLens Server

InfoClient

Getting Transform Map and Data Lens Information

Getting the data lens or Transform Map information uses a Java List interface. This is used to find out what data lenses or Transform Maps are available (deployed) on a particular server for processing. Details about the data lenses are also returned.

Import

Import the InfoClient with the following lines:

```
import oracle.pdq.api1.api.client.InfoClient;
import oracle.pdq.api1.api.client.ProjectData;
import oracle.pdq.api1.api.client.MapData;
```

Initialize the Client

For all the examples in this chapter, an instance of the `InfoClient` class needs to be created with the Oracle DataLens Server name and port.

The `SERVER_NAME` can be either a machine name such as "localhost" or an IP address "12.1.20.117".

The `SERVER_PORT` is the port number of the server. By default, the Oracle DataLens Server is installed on port 2229.

```
// Create the Server Api object and point to the server
InfoClient projectApi = new InfoClient(SERVER_NAME, SERVER_PORT);
```

Get a List of Deployed Data Lenses

With the `getDeployedProjectList` method, you get a List of ProjectData objects that contains all the Data Lenses that are **deployed and loaded** on the Oracle DataLens Server.

Each `ProjectData` object contains a:

- data lens name
- description of the data lens
- list of the Standardizations used by the data lens
- list of the Classification schemas used by the data lens
- list of the Unit Conversions used by the data lens
- single Source Locale used by the data lens
- list of the target Translation locales used by the data lens

This is a simple example of pulling the data from the returned List:

```
List prjList = infoApi.getDeployedProjectList();
Iterator itr = prjList.iterator();
        while (itr.hasNext()) {
            // Get the data lens information
            ProjectData prjData = (ProjectData)itr.next();
            String projectName =    prjData.getProject();
            String projectDesc =    prjData.getDescription();
            List standardizations = prjData.getStandardizations();
            List schemas =          prjData.getClassifications();
            List unitConversions =  prjData.getUnitConversions();
            String sourceLocale =   prjData.getSourceLocale();
            List targetLocales =    prjData.getTargetLocales();
        }
```

Lists of Schemas and Translations

The `ProjectData` object contains lists of classification and translation data for each data lens as shown above. These are just lists of String data. These lists include:

- Classification Schemas used (UNSPSC, eCl@ss, or user-defined)
- Target Translation locales supported.

In addition, the `ProjectData` object contains lists of input and output data for each data lens, as in the preceding. These are just lists of string data. These lists include:

- Input data list
- Output data list

Get a List of Deployed DSAs

With the `getDeployedWorkflowList` method, you get a List of `WorkflowData` objects that contains all the DSAs that are **deployed and loaded** on the Oracle DataLens Server.

Each `WorkflowData` object contains:

- DSA name
- Description of the DSA
- List of input fields
- List of output fields
- A list of Transform Maps used by this DSA
- A list of the Database Connections used by this DSA

Follwoing is a simple example of pulling the data from the returned list:

```
List workflowList = infoClientApi.getDeployedWorkflowList();
Iterator itr = workflowList.iterator();
while (itr.hasNext()) {
    // List the DSAs
    WorkflowData workflowData = (WorkflowData)itr.next();
    String name           = workflowData.getWorkflowName();
    String desc           = workflowData.getDescription();
    List inputFields      = workflowData.getInputFields();
    List outputFields     = workflowData.getOutputFields();
    List transformMaps    = workflowData.getTransformMaps();
    List dbConnections    = workflowData.getDbConnections();
}
```

4

Server Availability API to the Oracle DataLens Server

PingClient

This interface is the provided to access the availability of the Oracle DataLens Servers.

The `PingClient` class is used as an interface to the Oracle DataLens Server. This class provides a simple method that returns true if an Oracle DataLens Server is available for processing data and false if it is not.

Import

Import the `PingClient` with the following line:

```
import oracle.pdq.api1.api.client.PingClient;
```

Simple Server Check

This is a simple test to check the server availability. This can be used prior to sending data to the server for processing.

```
// Create the Server Api object and point to the server
PingClient pingApi = new PingClient(serverName, serverPort);
boolean available = pingApi.pingServer("JavaAPI");
```

Round-Robin Server Check

This is essentially the same code as in the previous section, except that you cycle through a list of Oracle DataLens Servers in a Production Oracle DataLens Server Group and returning the first server that responds.

```
Iterator iter = serverList.iterator();
while (iter.hasNext()) {
            String serverNameStr = (String)iter.next();
            if (new PingClient(serverName, serverPort).pingServer(userName)) {
                return serverNameStr;
            }
        }
```

5

Error Handling

This chapter describes the various error handling techniques.

Client-Side Exceptions

Client-side exceptions are caught via the standard java Exception catching mechanism. These faults are typically Connection exceptions, such as request/response timeouts or failure to connect to the server.

Client-Side Log Messages

The client side messages are output using standard output.

The client side messages can be output using a logging package called log4j. The API library uses Java reflection to determine if log4j is available. If it is already in your client application, log4j is used to output messages.

For more information about log4j, see the Apache log4j Web site:

http://logging.apache.org/log4j/

Log4j to Standard Output

By default, client-side error messages will go to standard output.

These same log messages are sent to a log-file if log4j is used in your client application.

If you want to add log4j for use in your client application, then the logger needs to be initialized. Otherwise, you will get the following error message in the standard output window or log file if there is a client-side problem.

```
log4j:WARN No appenders could be found for logger
(oracle.pdq.api1.api.client.ClientBase).
log4j:WARN Please initialize the log4j system properly.
```

Add the following line of code to initialize the client-side logger, enabling logging output to standard output.

```
import org.apache.log4j.BasicConfigurator;
. . .
// Initialize Log4J to get client-side logging to standard output
BasicConfigurator.configure();
```

The output in this log is usually a connection re-try attempt or some other client-side processing. Instead of the warning messages, you will now see the following types of messages:

```
- Attempt 2 to connect to http://lrivas-xp-a31:2229/datalens/Workflow
- Attempt 3 to connect to http://lrivas-xp-a31:2229/datalens/Workflow
- Attempt 4 to connect to http://lrivas-xp-a31:2229/datalens/Workflow
```

Log4J to a File

If log4j is being use in the client application, then the following will apply.

The client-side logging messages can be sent to a file as well. The following example is a very simple logging configuration that will log all the messages to a log file in the `/tmp` directory.

```
import org.apache.log4j.*;
. . .
// Initialize Log4J to get client-side logging to the /tmp directory
Logger logger =
Logger.getLogger(oracle.pdq.api1.api.client.ClientBase.class);
SimpleLayout layout = new SimpleLayout();
FileAppender appender = null;
try {
    appender = new FileAppender(layout,"/tmp/SCS_Log.txt",false);
} catch(Exception e) {}
logger.addAppender(appender);
logger.setLevel((Level) Level.WARN);
```

Note: The level for messages can be changed from `WARN` to `DEBUG` to get additional information if needed.

Server-Side Faults

There are also server-side exceptions that are propagated back to the client via the SOAP interface.

Here is how those exceptions are caught:

```
try {
m_client.getResultData (...
        } catch (Fault f) {
               System.out.println(f.getErrorCode());
        } catch (Exception e) {
            System.out.println("Error in Test: " + e.getMessage());
        }
```

Server-Side Exceptions

The Fault Exception object will provide a listing of error codes of the problem and status of your request to the Oracle DataLens Server.

Use the macros in the `oracle.pdq.api1.api.iface.ErrorIf` class to check for specific errors.

For example:

```
try {
resultData = wfgRtApi.getResultData(m_jobID, waitForResults);
    } catch (Fault f) {
        if (f.getErrorCode() == ErrorIF.ERROR_NOT_COMPLETED) ...
```

```
}
```

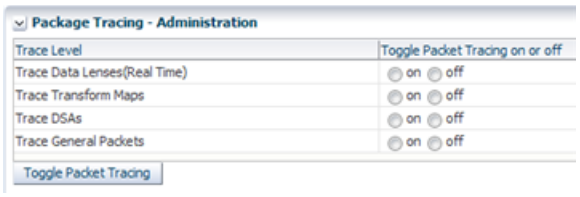
Server-Side Log Messages

Go to the Oracle DataLens Server Administration Web Pages and examine the log file from the home page. This will have a listing of any errors that were encountered in the server-side processing of your request.

Debugging Client Requests and Responses

The Oracle DataLens Server API communicates with the Oracle DataLens Servers by sending HTTP SOAP requests to the server and receiving HTTP SOAP responses back from the server. The content of these XML messages can be sent to standard output for debugging by the application programmer. This is useful if you want to verify that the data being send and received by the application program is exactly what is being communicated to the server.

You can enable this communication by logging into your Oracle DataLens Server Administration Web Pages and then clicking **Operations**. Select Trace Process Maps as in the following:



For more information, see *Oracle Enterprise Data Quality for Product Data Oracle DataLens Server Administration Guide*.

6

Compiling and Running with the API

Compile the Application with the Oracle DataLens Libraries

To compile your class with the Oracle DataLens Server client calls, the Oracle DataLens Server client libraries (`opdq-api1.jar` and `jdom-1.0.jar`) will need to be part of your `CLASSPATH`.

These jar files are located in the `\DevToolKit\java_api\lib` directory.

Add this to your `CLASSPATH` environment variable or use in the command-line Java compile as follows:

```
javac -classpath " opdq-api1.jar; jdom-1.0.jar" WfgClientTest.java
```

This creates the `WfgClientTest.class` file that is part of your application.

Run the Application with the Oracle DataLens Libraries

The Oracle DataLens Server libraries need to be referenced when running an application that accesses the Oracle DataLens Server.

The following is an example of running the program compiled in the previous example.

```
java  -cp " opdq-api1.jar; jdom-1.0.jar;.;" WfgClientTest
```

In this case, you are using the API and the `jdom-1.0` libraries, running the Java class file that were just compiled.

7

Customizing DSA Maps with Java Add-Ins and Algorithms

There are three types of customizations that can be done in a DSA and Transform Map (TMap).

- TMap Algorithms (The easiest to use)
- TMap Add-in Transforms
- DSA Add-in Output Adapters

The TMap Algorithms are the easiest to use and require no special work outside of the Application Studio. The Java code is written directly in the Transform Map Builder in Application Studio and can be tested and run from here. The limitation is that all the Java code needs to be part of the single transform method.

This Java code can contain entire packages of classes and needs to be written in an external Java API and imported into the Oracle DataLens Server.

TMap Algorithms

The following sections describe how to use the TMap Algorithms.

Initial Configuration

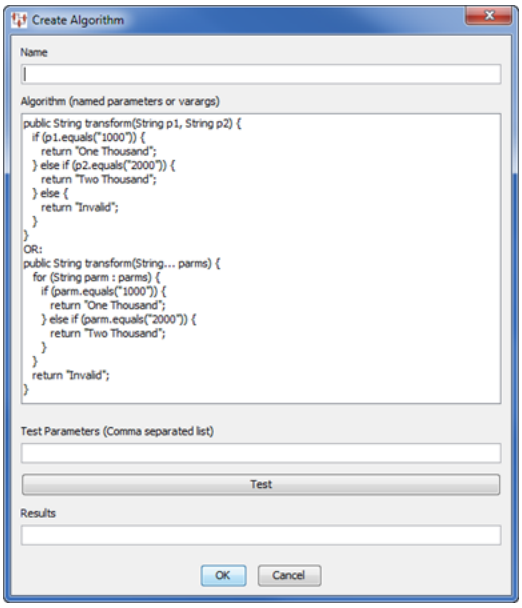
Java TMap Algorithms, allow Java code to be embedded into Transform Maps and have the code be executed when the parent DSA is run. The Oracle DataLens Client and Server software is configured ready-to-use to support this with no configuration changes needed.

Client Startup Changes

The Algorithm widget is available in the Process Control folder in the Graphical DSA Builder pane. See *Oracle Enterprise Data Quality for Product Data Application Studio Reference Guide*.

Creating a New TMap Algorithm

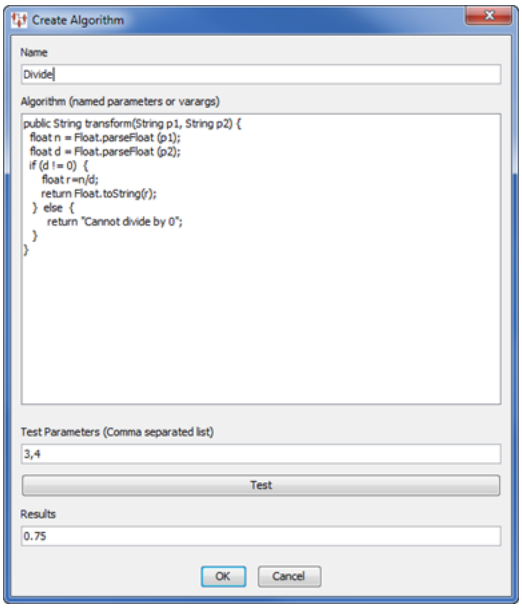
Drag the **Algorithm** widget into your Transform Map.



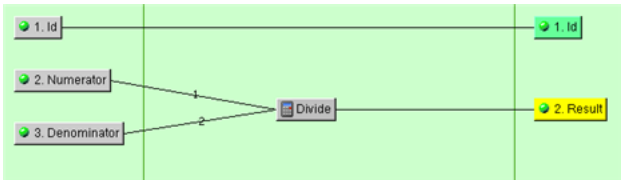
Notes on the Algorithm Java method:

- The name of the Algorithm can be any valid Java Class name, but the method needs to be declared as public String transform.
- The method may contain one or more string parameters. In the previous example, the method is taking two string parameters.

Next, name and create a new algorithm object with your algorithm. In this example, the template Java code was modified to divide two numbers and tested as in the following:



Click **OK** to save the new custom Algorithm. The Transform Map looks like the following:



The new Custom Algorithm is ready to check-in and deploy to the Oracle DataLens Server and start using in the client applications.

Transform Map Add-In Transforms

Java Transform Map Add-in Transforms allow user-defined widgets to be available for use in DSAs.

For additional information on these classes, see the file, `Add2Int.java` .

Writing a TMap Add-In Transform

The class may be in any Java package of your choosing.

The class name may be any valid Java Class name.

In the following example, you are using a Transform Map Add-in Transform class that is shipped with the Oracle DataLens Server installation called `GetField`.

The class must implement a constructor with a single string argument (the name).

```
public GetField(String name) {
    super(name);
}
```

The class must implement a method called `getResults` as follows:

```
/**
* This is the main method called by the Add-In Transform server code.
     *
     * @param linesOfInputData is an Array of data for each line being
    * processed,
     * where the data is an array of the inputs to a single computation.
     *
     * @param parameters are the input parameters for this TMap Add-in function.
     *
     * @return XfmInfo[] of the results of the transformation, one element
     * for each line of input data.
     */
  public XfmInfo[] getResults(String[][] linesOfInputData,
                                                  Map<String, String>
fixedParameters) {
        int inputLength = linesOfInputData.length;
        XfmInfo[] results = new XfmInfo[inputLength];
         // Get the parameters here…
        // Processing happens here…
        return results;
    }
```

1. `linesOfInputData` parameter - An array of arrays of Strings. The 1-D level array has one element for each line of input that must be processed. Each element of the 1-D array has a 2-D `String[]` containing the column data needed for the transformation. Thus the array looks like:

```
String[numberOfLines][numberOfColumnsOfInputData]
```

2. `fixedParameters` - These are the parameters to this add-in transform, passed in from the DSA.
3. Return an `XfmInfo[]` of the results of the transformation, one element for each line of input data.

Defining the Transform Map Add-In Transform

In the Application Studio, the add-in classes will be toggled on if the system finds the `AddInClasses.xml` file in the shared config directory on the Oracle DataLens Server.

Server

The class needs to be added to the `AddInClasses.xml` file, in the `C:\Oracle\Middleware\user_projects\domains\dls_domain\opdq\config`, directory, as follows:

```
<AddInClasses>
<Transforms>
      <class>
         <name>Get Field</name>

<className>com.onerealm.solx.maps.xfm.code.transform.GetField</className>
            <description>Gets the specified field from a string.  The field index,
            field separator, and default value are specified in the fixed
parameters.</description>
      </class>
   </Transforms>
</AddInClasses>
```

Defining the Input Parameters to the Transform Map Add-In Transform

This step can be skipped if the Transform Map Add-in transform does not use any initialization parameters.

The parameters to the new Transform Map add-in transform need to be added to the `AddInTramsformParameters.xml` configuration file. This file is located in the same configuration directory as the `AddInClasses.xml`. In this case, our `GetField` add-in takes three parameters and the `AddInTransformParameters.xml` file will look like similar to the following:

```
<TransformParameters>
<AddIn>
        <name>Get Field</name>
        <parameters>
            <parameter>
                <name>separator</name>
                <default>|</default>
                <desc>Separator for splitting string into fields</desc>
                <editable>true</editable>
            </parameter>
            <parameter>
                <name>index</name>
                <default>0</default>
                <desc>Index of field to extract (1-based)</desc>
                <editable>true</editable>
            </parameter>
            <parameter>
```

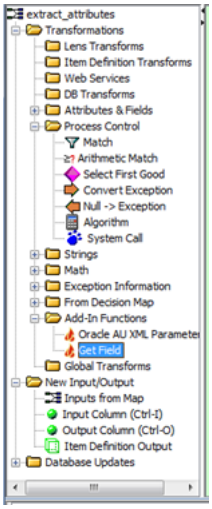
```
                <name>default</name>
                <default></default>
                <desc>Default value to return if field not found</desc>
                <editable>true</editable>
            </parameter>
        </parameters>
    </AddIn>
</TransformParameters>
```

Using the Transform Map Add-In Transform in the Client

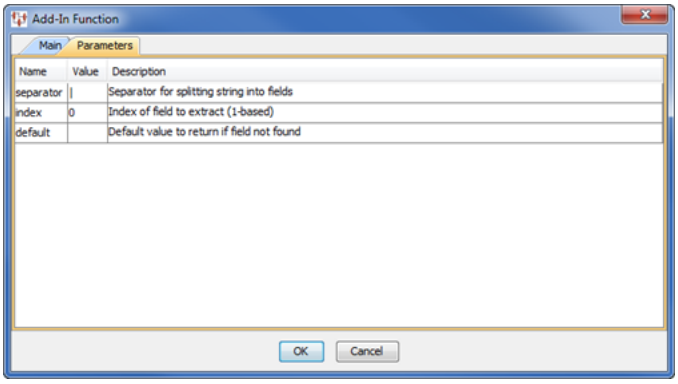
No changes are need for the client configuration to pick up your new Tmap Add-In Transformation. The Oracle DataLens Server just needs to be restarted whenever the `CustomClasses.xml` file is updated (because the server reads this file on startup).

> **Note:** You will be able to add the new Tmap Add-in to your DSA map, but this cannot be tested on the client, it can only be tested by running a job on the server.

Start the Application Studio. The new add-in class is available in the Transform Map as follows:



Drag the **Get Field** widget into the Transform Map, then click the **Parameters** tab so that the parameters are displayed for you to edit as follows:



Using TMap Add-in Transforms to Process Exception Data

The individual Tmaps that comprise the complete DSA, have a limitation in that the same number of records input to the Tmap must be output from the Tmap. If the Tmap is processing data and some of the records cannot be processed, then these individual records must be flagged as exception records and routed in the DSA for separate processing.

You can flag exception records to be processed separately using the `XfmError` class as in the following example:

```
if (returnUnmatchedRows) {
   // This is a good row of data
   finalResults[i] = new XfmData(classification.getId()+"|0|||||", 0);
} else {
   // Return a line failure (exception)  for this line
   finalResults[i] = new XfmError("unknownMap", getName(), "No Match");
}
```

The `XfmError` constructor uses the following three parameters that are all string values:

`mapName`
The name of the DSA.

`xfmName`
The name of the Tmap transform.

`errorMsg`
The error message associated with the line of data.

DSA Add-In Output Adapters

Java DSA Add-in Output Adapters can write data out in any user-defined format. Since this is an output step, there is no routing of data past this step in the DSA Map, so this should be used only by Java code that will not be throwing any exceptions that need to be trapped and processed by the DSA Map.

Writing a DSA Add-In Output Adapters

The class may be in any Java package of your choosing.

The class name may be any valid Java Class name.

In the following example, you are using a Transform Map add-in transform class that is shipped with the Oracle DataLens Server installation called **SCS XML**.

The main method that is called is **`writeOutput`**. This returns a `WfgCustomOutputReturn` object, which contains information needed to forward the result data to an email address or an FTP site. If this returns null, then there will be no email or FTP.

> **Note:** Even if this object is returned, the email and FTP is only sent if it is defined in the DSA or the DSA job has defined email or FTP output.

Following is an example of the structure of the Output Adapter Class:

```
package com.onerealm.solx.maps.wfg.code.output;
```

```
    /**
     * @param job  PMap job information
     * @param data Wfg Job data
     * @param outputDir The directory to write the xml file
     * @param parameters The input parameters to the Add-in Output Adapters.
     * @return Custom output
     * @throws SaException Superclass for all SCS exceptions
     * @throws IOException Signals that an I/O exception of some sort has occurred
     */
public WfgCustomOutputReturn writeOutput(WfgJob job, WfgInputData data, String
outputDir,
                                         Map<String, String> parameters)    {
        List<WfgDataLine> lines;
        while ((lines =
data.getNextGoodLines(WfgConstants.MAX_MEMORY_LINES)) != null) {
            for (WfgDataLine line : lines) {
                System.out.println(line.getData());
            }
        }
        return null;
    }
```

Defining the DSA Add-In Output Adapter

In the Application Studio, the add-in classes will be toggled on if the system finds the `AddInClasses.xml` file in the `shared/config` directory on the Oracle DataLens Server.

Server

The class needs to be added to the `AddInClasses.xml` file as follows:

```
<AddInClasses>
   <Outputs>
        <class>
            <name>SCS XML</name>

<className>com.onerealm.solx.maps.wfg.code.output.scspim.ScsStepPimProducts</class
Name>
             <description>This will output a STEP PIM Product XML document
with SCS processed data; The default file is
/tmp/ScsStepPimProductData_jobId.xml</description>
        </class>
  </Outputs>
</AddInClasses>
```

> **Note:** There is *no* client file that needs to be updated for use with the Application Studio.

The server file needs to be updated for use running DSAs on the Oracle DataLens Server and to make this available to the Application Studio Clients.

Follow the instructions in the Transform Map Add-in Transforms section for changing the startup scripts and adding the new classes to the classpath.

Defining the Input Parameters to the Transform Map Add-In Transform

This step can be skipped if the Transform Map Add-in transform does not use any initialization parameters. In this example, you are not using any input parameters.

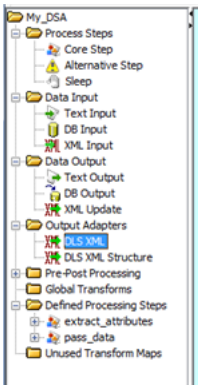
Using the DSA Add-In Output Adapter in the Client

No changes are need for the client configuration to use your new DSA Add-In Output Adapter. The Oracle DataLens Server just needs to be restarted whenever the `CustomClasses.xml` file is updated (because the server reads this file on startup).

Note: You will be able to add the new DSA Add-in Output Adapter to your DSA map, but this cannot be tested on the client, it can only be tested by running a job on the server.

Use in the Application Studio

Once all of the steps in the preceding sections are completed, the new DSA Output Adapter is now available for use in the Application Studio as follows:



A

Working Through a Proxy Server

Sometimes a Java program needs to call the Oracle DataLens Java API to an Oracle DataLens Server outside of your firewall. Normally this is not a problem, but sometimes there is a proxy server that must be negotiated to get to the outside world.

There are two solutions to this problem.

- Use the Oracle DataLens Web Services interface and you your WSDL connection software to negotiate through the proxy server.
- Use the Java Proxy arguments to the `java` command.

Run-Time Java Proxy Parameters

Three parameters are used with the `java` command to set the proxy information:

- `DproxySet=true`
- `DproxyHost=`*hostname or IP Address*
- `DproxyPort=8080`

These parameters are illustrated in the following example `java` call to a program called `WfgProgram`:

```
java -cp "./opdq-api1.jar;./jdom-1.0.jar;." -DproxySet=true
-DproxyHost=10.1.60.116 -DproxyPort=8080 WfgProgram
```

RtClient Java Proxy Parameters

There are four additional parameters to the RtClient overloaded constructor for with through a proxy. For additional information, see the JavaDoc documentation as described in "Related Documents" on page 2-vii.

- `@param proxyHost` - the name of the proxy server
- `@param proxyPort` - the port of the proxy server
- `@param proxyUser` - the user name to login to the proxy server
- `@param proxyPassword` - the password to login to the proxy server

These are shown in the following between the `ServerPort` and the `ENCRYPTION` parameters:

```
m_wfgClient = new WfgClient(serverName, serverPort,
              "10.1.60.106",2229, "user123", "secret1",
                          ENCRYPTION, clientCode, APPLICATION);
```