opentext™

OpenText™ Embedded Data Transformation Engine

Developer's Guide

This guide is intended for Data Transformation Engine developers and provides information on developing with APIs and other more intermediate features.

VDTOTS240200-PDT-EN-1

OpenText™ Embedded Data Transformation Engine
Developer's Guide
VDTOTS240200-PDT-EN-1
Rev.: 2024-Apr-16

This documentation has been created for OpenText™ Embedded Data Transformation Engine CE 24.2.
It is also valid for subsequent software releases unless OpenText has made newer documentation available with the product, on an OpenText website, or by any other means.

Open Text Corporation

275 Frank Tompa Drive, Waterloo, Ontario, Canada, N2L 0A1

Tel: +1-519-888-7111
Toll Free Canada/USA: 1-800-499-6544 International: +800-4996-5440
Fax: +1-519-888-0677
Support: https://support.opentext.com
For more information, visit https://www.opentext.com

Table of Contents

Chapter 1

Introduction

This guide contains the information needed to access the OpenText Embedded Data Transformation Engine base engine through the Java Application Programming Interface (API).

Note: This guide assumes the reader's familiarity with the use of the Java programming language, as well as with Data Transformation Engine. Please refer to *OpenText Embedded Data Transformation Engine User Guide* for help with the use of the product.

1.1 Getting Started

Data Transformation Engine consists of two primary components: the OpenText Output Transformation Server and the run-time engine. OpenText Output Transformation Designer is used to create the template file for your transformations. Once you have a template, you may use the engine to perform your actual transformations.

Note: A new template is required for each transformation that requires different configurations; for example: different input format, output format, and output values.

1.2 About the Java API

The application's engine can be embedded in your own application via the Java Data Transformation Engine Programmatic API. The API supports configuration and transformation functions as specified in the *OpenText Embedded Data Transformation Engine Javadoc*, which can be accessed from the OpenText Output Transformation Suite help system.

Note: Data Transformation Engine supports engine reusability. You may now reuse the `MagicGateTransformer` object, rather than create multiple instances of the engine, in order to perform as many transformations as needed.

Examples of using the API, including handling Data Transformation Engine exceptions, are located in the `<install_home>\initialFiles\common\_sample\DataTransformation\embed_sample` directory. These samples can be used as a reference for your own code.

Chapter 2

Setting Up a Transformation

There are five mandatory steps for performing transformations in your Java code:

1. Add import statements.
2. Create a StreamSource object from your template file.
3. Create a `MagicGateTransformFactory` object.
4. Create a `MagicGateTransformer` object (the engine).
5. Call the transform method.

As well, you may use any of the other public methods of the `MagicGateTransformer` interface as described in *OpenText Embedded Data Transformation Engine Javadoc* for transformation configuration purposes.

2.1 Import Statements

In order to have access to the classes you will need in performing transformations, the following import statements are required in your code:

```
import com.xenos.transform.*;
```

```
import javax.xml.transform.stream.StreamSource;
```

These are in addition to any other import statements your code may need.

2.2 Template StreamSource Instantiation

In order to configure the engine for use with a particular template file, you will need a `StreamSource` object created from your template file. The StreamSource class has many constructors, but only one will be described here. See *OpenText Embedded Data Transformation Engine Javadoc* for more detail.

The easiest way to create your `StreamSource` object is via the constructor that takes a file object as its only parameter. For example, if `template` is a String object whose value is the location of your template file on your system, your `StreamSource` object instantiation should resemble the following:

```
StreamSource templateSource = new StreamSource(new File(template));
```

where `templateSource` will be the name of your `StreamSource` object.

2.3 MagicGateTransformFactory Instantiation

The `MagicGateTransformFactory` is the class used to generate the `MagicGateTransformer` object that will act as your base engine. The public static method `newInstance()` is used in order to obtain a new `MagicGateTransformFactory` instance.

```
MagicGateTransformFactory mgtFactory = MagicGateTransformFactory.newInstance();
```

This code creates a `MagicGateTransformFactory` object named `mgtFactory`.

2.4 MagicGateTransformer Instantiation and Configuration

The `MagicGateTransformer` object you create will act as your base engine. Since `MagicGateTransformer` is an interface and not an actual class itself, objects of this type must be created using the `newTransformer(StreamSource templateSource)` method of your `MagicGateTransformFactory` object.

Supposing you already have a `StreamSource` object called `templateSource`, and a `MagicGateTransformFactory` object named `mgtFactory`, the following code would generate your `MagicGateTransformer` object, `mgtTransformer`:

```
MagicGateTransformer mgtTransformer = mgtFactory.newTransformer(templateSource);
```

2.5 Configuring the Engine

Once you have your `MagicGateTransformer` object, you may call any of its public methods as described in the *OpenText Embedded Data Transformation Engine Javadoc*. All but the transform method (described in the next section) are optional in your code.

2.5.1 Variables

There are several methods provided for dealing with variables you may have defined during template creation in Output Transformation Designer. You are permitted to use variables in filenames, output item values, and most other user-defined values for your transformation configuration. Data Transformation Engine allows you to nest variables, that is, to use the value of a variable within the value of another variable. See the variables sections in the *OpenText Embedded Output Transformation Engine User Guide* for information on defining your variables in Output Transformation Designer.

Variables are adjustable at run-time using public methods belonging to the `MagicGateTransformer` interface. See *OpenText Embedded Data Transformation Engine Javadoc* for help with the getProperty, setProperties and setProperty methods.

2.5.2 Engine Reusability

Your `MagicGateTransformer` object may be reused to perform multiple transformations rather than having to create multiple instances of this interface.

If multiple threads are used in your application, each thread must have its own `MagicGateTransformer` object.

2.6 Marshalling Java Objects

Java objects can be converted to XML strings using the marshallObject method of the Data Transformation Engine API. For more detailed information on the use of the marshallObject, see *OpenText Embedded Data Transformation Engine Javadoc*, located in the OpenText Output Transformation Suite help system.

2.7 Transform Methods

Now all that remains is calling the transform method on the engine. See one of the topics in this section, depending on your type of transformation.

Note: All types of transformations deal with the `javax.xml.transform` package. For more information, see "javax.xml.transform Package" on page 13.

2.7.1 JDBC Input

If your transformation involves JDBC input, the transform method you will use takes a single parameter; a Result object defined for your output source. The code should resemble the following:

```
mgtTransformer.transform(new StreamResult(new FileWriter(outputFilename)));
```

where `outputFilename` contains the desired file name of the output file. Also, the `StreamResult` class permits construction via OutputStream, Writer and String objects.

2.7.2 JDBC Output

If your transformation involves JDBC output, the transform method you will use takes a single parameter: a Source object defined by your input source. Your code should resemble the following:

```
mgtTransformer.transform(new StreamSource(new File(inputFilename)));
```

where `inputFilename` contains the filename of the input source. Your Source object does not necessarily have to be created from file. The `StreamSource` class also has constructors that use InputStream, Reader and String objects as their parameter.

2.7.3 Single Target

If your transformation writes to a single target destination, the transform method you will use takes the following two arguments:

- A Source object defined by your input source.
- A Result object defined by your target destination.

Your code should resemble the following:

```
 mgtTransformer.transform(new StreamSource(new File(inputFilename)), new
StreamResult(new FileWriter(outputFilename)));
```

where `inputFilename` and `outputFilename` are String objects that contain the filenames of the input and output, respectively. Your Source and Result objects do not necessarily have to be created from files. The `StreamSource` class also has constructors that use InputStream, Reader and String objects as their parameter. Additionally, the `StreamResult` class permits construction via OutputStream, Writer and String objects.

Note: As a reference, see the SimpleTransform.java example in the `<install_home>\initialFiles\common\_sample\DataTransformation\embed_sample` directory.

2.7.4 Multiple Inputs

If your transformation involves multiple inputs, the transform method you will use takes the following two parameters:

- A Map object with the keys being the input names defined in the mapping with Source objects as the objects.
- A Result object defined by your target destination.

The code should resemble the following:

```
Map input = new HashMap()
inputMap put(“INPUT 1”, new StreamSource(new File(inputFileName1)));
inputMap put(“INPUT 2”, new StreamSource(newFile(inputfileName2)))
mgTransformerTransform(inputMap, new StreamResult(new FileWriter(outputFileName));
```

where `INPUT 1` and `INPUT 2` are your input names in the mapping, `inputFileName1` and `inputFileName2` are your input files, and `outputFileName` is your target result.

2.7.5 Multiple Targets

If your transformation has multiple target destinations, the transform method you will use takes the following two arguments:

- A Source object defined by your input source.
- A Map object defined by your target destinations.

Your code should resemble the following:

```
mgtTransformer.transform(new StreamSource(new File(inputFilename)), outputMap);
```

where `inputFilename` is the String object whose value is your input filename. Your Source object does not necessarily have to be created from file. The `StreamSource` class also has constructors that use InputStream, Reader and String objects as their parameter.

The `outputMap` argument is a java.util.Map object whose keys are the IDs of each output target as listed in your template file, and whose values are `javax.xml.transform.Result` objects containing your output destinations.

Note: As a reference, see the Multiple Results Java code example in the `<install_home>\initialFiles\common\_sample\DataTransformation\embed_sample` directory.

2.7.6 Single Target with Multiple Outputs

If your transformation has multiple output results from a single target, the transform method you will use takes the following two arguments:

- Source object defined by your input source.
- MultiResults object defined by your target destination.

Your code should resemble the following:

```
mgtTransformer.transform(new StreamSource(new File(inputFilename)), new
MultiResults(new File(output)));
```

where `inputFilename` is the String object whose value is your input filename. Your Source object does not necessarily have to be created from file. The `StreamSource` class also has constructors that use InputStream, Reader and String objects as their parameter.

The MultiResults output object will generate multiple output files with a sequence number appended to the end of each filename. This is the case for all output data formats with the exception of PDF. If a MultiResults object is used to create multiple PDF files, and if the /PDFRoot@OUTPUT_ID attribute in the template is assigned a value, then Data Transformation Engine will use this value as the filename without generating a sequence number. If no value is assigned, Data Transformation Engine will sequence the output.

Note: As a reference, see the Single Target Multi Results Java code example in the `<install_home>\initialFiles\common\_sample\DataTransformation\embed_sample` directory.

Chapter 3

javax.xml.transform Package

The `javax.xml.transform` package contains APIs used for managing and performing transformations. When embedding a Data Transformation Engine transformation into your own application, you are required to use classes implementing the `javax.xml.transform.Source` and `javax.xml.transform.Result` interfaces as a means of defining your input source(s) and output destination(s).

The **Source** interface has four implementing classes available for your use:

- **javax.xml.transform.dom.DOMSource** - Holds your source in the form of a Document Object Model tree.
- **javax.xml.transform.sax.SAXSource** - Holds any SAX-style source.
- **javax.xml.transform.stream.StreamSource** - Holds your source in the form of a stream of XML markup.
- **com.xenos.transform.ObjectSource** - Holds your source in the form of a

  ```
  java.lang.Object
  ```

The **Result** interface has five implementing classes available for your use:

- **javax.xml.transform.dom.DOMResult** - Holds your result in the form of a Document Object Model tree.
- **javax.xml.transform.sax.SAXResult** - Holds your SAX-style result.
- **javax.xml.transform.stream.StreamResult** - Holds your result in the form of a stream of XML markup.
- **com.xenos.transform.ObjectResult** - Holds your result in the form of a

  ```
  java.lang.Object
  ```

- **com.xenos.transform.MultiResults** - If you have multiple outputs for a single target, your results can be placed in this array.

3.1 Pipeline Transformations

Careful use of the classes mentioned previously can simplify the pipeline transformations you may perform. Pipeline transformations are transformations from an input source or sources to an intermediate result, and then into your final desired destination(s). For example, you may first transform your input into a DOMResult object. By using the `getNode()` method of the DOMResult object, you can easily create a DOMSource object from which to perform another transformation. At no point was it necessary to save the intermediate result to disk.

Refer to the PipeTransform.java example in the `<install_home>\initialFiles\common\_sample\DataTransformation\embed_sample` directory for help with pipeline transformations. For information about any of the classes mentioned above, see the corresponding JDK1.4 API documentation, or *OpenText Embedded Data Transformation Engine Javadoc*.

Chapter 4

Transformations in Multiple Thread Environments

You may find it useful to run several transformations in separate threads at the same time. This is easily accomplished by embedding the transformation process into the `run()` method of a class you create extending the Thread class.

Refer to the `MultipleThreadsTransform.java` sample in the `<install_home>\initialFiles\common\_sample\DataTransformation\embed_sample` directory for an example.

Note: For more information on embedding the transformation process, see "Setting Up a Transformation" on page 7.

4.1 XML to Multiple Database

Setting up a transformation for multiple database output is the same as setting up any multiple output transformation. In Output Transformation Designer, open the **Settings and Details window**. At the bottom of the pane is the **Insert a Tab** button.

The number of tabs you have in the **Target** pane corresponds to the number of databases to which you will be sending output. Insert as many as necessary. Configure your output for each database as normal in each respective tab.

4.2 Multiple Database to XML

Transformations involving multiple input databases are possible through the use of the JDBC lookup feature. The database from which you will be drawing most of your input data you will set as your input source. For the output items that require data from other databases or other database tables, you will define JDBC replace functions to extract the desired data. When creating your template in Output Transformation Designer, do the following:

1. Define all necessary JDBC connections in the **JDBC Connection** tab of the Settings and Details window.

2. Go to the **JDBC Lookup Table** tab in the Settings and Details window. For each additional database table you use, you will need a new lookup tab configured to a different JDBC connection. The value you enter in the **Replace With** field is the name of the column from which you will be retrieving data. The value that goes in the **Find What** field is the value you must give as an argument in your replace function.

3. Configure the output items that will be taking values from your primary database as usual.

4. Output items that will be taking values from one of the other database tables must be mapped to one of the replace functions you have defined.

For more detailed information on JDBC replace functions, see *OpenText Embedded Data Transformation Engine - User Guide (VDTOTS-H-UDT)* and *OpenText Embedded Data Transformation Engine - User Guide (VDTOTS-H-UDT)*.

Chapter 5

JNDI Lookup

Data Transformation Engine offers a JNDI lookup feature. When you connect to a database using the JNDI Connection option in Data Transformation Engine, you are able to run several applications within the same JEE server.

Chapter 6

Function Naming Conflict

If you are running multiple instances of Data Transformation Engine within your JVM, you may have to watch for conflicting user-defined function names.

For example, suppose you have a custom function called `myCustomFunction` in your first instance of Data Transformation Engine, and a custom function called `myCustomFunction` that has different functionality in your second instance. With the two being run in the same JVM, only the second `myCustomFunction` will be used in both instances of Data Transformation Engine.

This naming conflict occurs in:

- Replace functions
- JDBC replace functions
- Custom functions
- JDBC lookup IDs

Chapter 7

Output Transformation Server Classpath and License

7.1 Output Transformation Server Classpath

The Output Transformation Server classpath is where you load the JAR files to use Output Transformation Designer and ultimately Data Transformation Engine from. The JAR files are loaded from `<install_home>\lib\common` (`%ES_HOME%\lib\common` from the command prompt), which is set in `\dev-studio\bin\DeveloperStudio.bootstrap` under the `-libDirs` property.

For more information about the Output Transformation Server Classpath, see *OpenText Output Transformation Server - Developer's Guide (VDTOTS-H-PGD)* in *OpenText Output Transformation Server Developer's Guide*.

In addition to the above, any compiled classes can be placed under

`<install_home>\initialFiles\common\_classes`

7.2 Output Transformation Server License Key

In order to use Data Transformation Engine, you must provide a license key. The easiest way to do this is to type your serial key into a text file and name it `serialnum.txt`. Place this file into your Data Transformation Engine root directory, or from your JVM booting directory. Alternatively, you may type your serial key into a text file and give it any name and location you wish, provided you call the `setLicenseFile (java.io.File` license) method of your `MagicGateTransformFactory` object with the correct File argument.