opentext™

OpenText™ Embedded Output Transformation Engine

Developer's Guide

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

VDTOTS240200-PTE-EN-1

OpenText™ Embedded Output Transformation Engine
Developer's Guide

VDTOTS240200-PTE-EN-1
Rev.: 2024-Apr-16

This documentation has been created for OpenText™ Embedded Output 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

© 2024 Open Text

Patents may cover this product, see https://www.opentext.com/patents.

Disclaimer

No Warranties and Limitation of Liability

Every effort has been made to ensure the accuracy of the features and techniques presented in this publication. However, Open Text Corporation and its affiliates accept no responsibility and offer no warranty whether expressed or implied, for the accuracy of this publication.

Table of Contents

Chapter 1

Introduction

1.1 API Objective

This guide has been created to benefit developers who wish to embed the functionality of OpenText Embedded Output Transformation Engine into a Java application. It is assumed that any developer reading this document is familiar with the product, the standard configuration process, the Output Transformation Engine API, and the Java programming language.

The API allows Java developers to:

- Have a flexible and highly customizable interface to the application's engine, as well as several optional features for the developer to customize the workflow of data into, through, and out of the engine during runtime.
- Customize existing built-in components with pure Java integration. Developers can extend Output Transformation Engine classes to provide additional data processing functionality within their applications.
- Create custom components that receive data from a parser and give the developer access to the parsed input data along with data extracted from fields. These custom components, or user extensions, can be developed either by customers or by our Professional Services consultants.
- Have complete access to page information. Developers can inspect all data that is passed from a parser to the generators, as well as the data passed from the built-in Output Transformation Engine components.

1.1.1 Overview of Packages

These packages are available to use with the application.

com.xenos.d2e

This package contains a fundamental class, d2eProcess, which is needed to control the base engine. d2eProcess is based on a singleton model so that multiple jobs can be submitted to a single running engine. This package also contains the JobRecord class that contains the information about a single job that can be submitted to the engine.

com.xenos.d2e.exceptions

This package contains the exception classes that may be thrown from within the engine. These exceptions may contain nested exceptions containing the exceptions that caused this one to be thrown. These exception classes have been created from the ground up such that Java 1.4 is not required for nested exceptions. The most

common exceptions to be thrown are `XenosException`, `XenosIOException` and `XenosRuntimeException`.

com.xenos.d2e.log

This package contains classes used to receive log messages as well as to send messages to the log. There are two types of logs to subscribe to: job logs and system logs. You need to implement the `IJobLogSubscriber` interface to receive job level log messages and implement the `ISystemLogSubscriber` interface to receive system level log messages. To be able to handle messages through your own `LogMessageHandler`, you will need to extend the `LogMessageHandler` class as well.

Chapter 2

Developing in Output Transformation Engine

Development of custom components in Output Transformation Engine can help you adapt projects for your specific needs. For more information, consult the following topics to get started with developing in Output Transformation Engine.

2.1 Javadocs

The *Javadoc* document provides a list of all Java packages and their associated class/interface descriptions. This document is an invaluable resource when developing your own components and is available from the **Help** menu in Output Transformation Designer.

2.2 Sample code

Various code samples of projects and components are included with the application. Sample code can be found in the following location:

- `<install_home>\install\<version>\initialFiles\common\_sample\OutputTransformation\applications\sample`

Java API samples are also bundled with the application. For more information on Java API samples, see "Output Transformation Engine Java API Sample Code" on page 11.

Chapter 3

Output Transformation Engine Java API Sample Code

Java API samples are available to exhibit certain concepts covered in this guide and also demonstrate how to create custom Java components for use in your projects. The Java API samples can be found in the following location:

- `<install_home>\BaseRepositories\designer\designer\common\_sample\OutputTransformation\api`

A few essential API samples are described in this chapter. Additionally, some folders contain ReadMe files with further information.

3.1 SampleMain

The `SampleMain` program is designed to be launched from the command line with the standard arguments for Output Transformation Engine. It demonstrates how to set up the start up parameters, set job variables, and submit the job.

3.2 SampleRVC

This class shows how to create a Resource Version Control (RVC) component to perform resource versioning. For more information, see "Resource Version Control" on page 35

3.3 BufferedSimpleBranch

This class demonstrates a very simple User Extension custom component that extends the `BufferedBaseComponent` class.

3.4 SampleUserComponent

This class demonstrates various aspects of User Extensions custom components. It shows how to extract information from XFT events along with accessing application parameters. The directory also contains the `components.xml` file that identifies the `SampleUserComponent` to Output Transformation Designer.

3.5 InsertImagesComponent

This class illustrates how the `XifElementUtil` class may be used to insert images onto a XifPage. It is recommended that you initially use this component with a PDF Generator, as PDF output is typically the easiest to test and also supports a large set of image features that other output formats do not.

Chapter 4

Starting and Controlling the Output Transformation Engine Base Engine

A core design concept of the Output Transformation Engine base engine is the notion of a job. A **job** is an instance of the application which is submitted to the engine to be processed.

4.1 Starting the Base Engine

To start the base engine:

1. To create a job, you must first get a reference to the **D2eProcess** object. You will need to import two packages:

```
 import  com.xenos.d2e.*;
import  com.xenos.d2e.exceptions.*;
```

   Since the d2eProcess object is implemented as a singleton, you can get the reference by calling:

```
m_d2eProcess = D2eProcess.getInstance();
```

2. Set your Output Transformation Engine system configuration file (.d2esys file):

```
m_d2eProcess.setSystemConfig(D2eSysFile);
```

3. Start your system:

```
try {
m_d2eProcess.startSystem();
}
catch (SystemException se) {
System.out.println(se.getDetails());
System.exit(se.getExitCode());
}
```

   Note: Refer to *OpenText Embedded Output Transformation Engine User Guide* and *OpenText Embedded Output Transformation Engine Javadoc* for additional parameters.

   The engine is now running and you are ready to start creating jobs.

4.2 Creating Jobs

To create a job, you must create a new job and submit it. In order to create a job, you need to have already started the engine. For more information, see "Starting the Base Engine" on page 13

```
// Get the D2eProcess instance
D2eProcess d2eProcess = D2eProcess.getInstance();
// Create a new JobRecord
try {
JobRecord jobRec = d2eProcess.createJob(applicationName,  inputFileName,
outputFileName);
} catch (SystemException se) {
throw se;
}
```

The **D2eProcess.createJob** method is overloaded as follows:

```
createJob(String applicationName)
createJob(String applicationName, InputStream is, OutputStream  os)
createJob(String applicationName, String inputFile, String  outputFile)
```

where:

*@param `applicationName` is either the full application (project) path, or the logical application (project) name.

If you are passing in input and output streams, then:

- `*@param is` is the input stream that `IoUtility` will retrieve via the `JobRecord`
- `*@param os` is the output stream that `IoUtility` will retrieve via the `JobRecord`

If you are passing in input and output files, then:

- `*@param inputFile` corresponds to the location in a file definition parameter
- `*@param outputFile` corresponds to the location in a file definition parameter

The D2eProcess.createJob method can throw a SystemException if an error occurs when loading the application. This exception must be caught.

The D2eProcess.createJob method returns a new job record with a parsed XdcApplication.

4.3 Submitting Jobs

After you have created your job, you are ready to submit it to the engine for processing. To do this you need to get the JobManager from the Output Transformation Engine process.

```
JobManager jobManager = m_d2eProcess.getJobManager();
```

After getting the JobManager, you must submit the job to it:

```
jobManager.submitJob(jobRecord, true);
```

You have the option of waiting for the job to complete or not. If you set the second argument to **true**, the method will not return until the job has completed. If you set it to **false**, the method will return automatically without waiting for the job to complete.

4.4 Programmatically Modifying Parameters

Project parameters can be modified through the API. Projects may be cached, so care must be taken when modifying any parameters in any system where two or more jobs are run concurrently. In this situation you must clone the application to be modified.

1. Create a clone of the **JobRecord XdcApplication**. This class holds all of the component parameters.

   ```
   XdcApplication cloneApp = (XdcApplication)jobRecord.getApplication().clone();
   ```

2. Get a reference to the particular parameter class of interest. In this example we want the AFP Parser parameters. This component is called `parser1` within the project file.

   ```
   AfpParserParm afpParm = (AfpParserParm)cloneApp.getComponetParmByName("parser1");
   ```

3. Get a reference to the AFP Parser parameters page size parameter.

   ```
   PageSizeParm pageSize = (PageSizeParm)afpParm.getDefaultPageSize();
   ```

4. Set to the desired page size in the PageSizeParm.

   ```
   pageSize.setXSize((float)5.0);
   pageSize.setYSize((float)5.0);
   ```

5. Set the cloned XdcApplication back into the JobRecord.

   ```
   jobRecord.setApplication(cloneApp);
   ```

6. Submit this JobRecord to the JobManager.

   ```
   jobManager.submitJob(jobRecord, true);
   ```

4.5 Job Variables

Job variables are used to provide dynamic substitutions within projects. To use a job variable, you must define a job variable as follows:

```
 jobRecord = m_d2eProcess.createJob(applicationName);
 ...
 jobRecord.getApplication().getJobVariables().setProperty("afpFontVar",
"{d2eAppPath}afpfonts");
```

Note: Be careful when referencing job variables within job variables, as you may cause circular references.

Then, you can use this job variable in your project in place of a file definition or wherever name substitutions occur.

Job variables can be added programmatically to the job as mentioned above or they can be defined first in the `.d2eproj` file and modified programmatically using the statement mentioned above. You may add the job variable to the `.d2eproj` file by adding the following line:

```
 <application name="myapplication-afp2pdf"  cache="Job">
 <description> myapplication-afp2pdf </description>
 <jobvariable name="afpFontVar" value="{d2eAppPath}afpfonts"  ispath="true"/>
```

The **ispath** attribute specifies whether this job variable represents a path or not. If you set ispath="true" then you can reference your job variable as follows:

```
 <filedeflist name="FdAfpfonts">
 <filedef directives="{afp}" location="{afpFontVar}{0}.fnt"  ioHandlerFactoryInstance=""
cache="job"/>
 </filedeflist>
```

Otherwise, you would need to use the following:

```
 <filedeflist name="FdAfpfonts">
 <filedef directives="{afp}" location="{afpFontVar}/{0}.fnt"
ioHandlerFactoryInstance="" cache="job"/>
 </filedeflist>
```

Note: A filedeflist can contain one of more filedef objects. A filedef object encapsulates information about the source or destination of a data stream.

The example above illustrates that by programmatically setting your job variable, you can change the path to your ACIF resource file, on a per job basis.

4.6 Inspecting Job Results

After submitting a `JobRecord` to the Output Transformation Engine `JobManager` for execution, you will probably wish to programmatically inspect the results of the job. When a job has finished, the `JobRecord` object will contain attributes such as a completion code, error messages and warning messages.

You can submit a job using this method:

```
jobManager.submitJob(jobRecord, true);
```

By specifying the second argument value as **true**, you are assured that when the `submitJob()` method returns, it means the job has finished executing. For more information, see "Submitting Jobs" on page 15.

However, there are several possible conditions that may cause a job to finish, including:

- Ran successfully without any errors.
- Failed to initialize due to a configuration error.

This is recorded in the `JobRecord` as the **completion code**. Your business logic may require you to identify the completion code and take action depending on the actual value of the completion code.

The class `com.xenos.d2e.CompletionCode` is used to represent the completion code for a job. To obtain the CompletionCode from the JobRecord, perform the following method call:

```
CompletionCode cc = jobRecord.getCompletionCode();
```

As listed in *OpenText Embedded Output Transformation Engine Javadoc*, the `CompletionCode` class has the following constant enumerations:

Table 4-1: Completion Code Values

Completion Code Value	Task_ID	Task_Description
0	TASK_SUCCESSFUL_ID	The task has completed successfully without errors.
4	TASK_SUCCESSFUL_WITH_WARNINGS_ID	The task has completed successfully, but one or more warning conditions have been met.
8	TASK_FAILED_WITH_ERRORS_ID	The task has failed because one or more error conditions have been met.

Completion Code Value	Task_ID	Task_Description
12	TASK_FAILED_WITH_RUNTIME_EXCEPTIONS_ID	The task has failed because one or more unexpected runtime exceptions have occurred.
16	TASK_FAILED_ON_INITIALIZATION_ID	The task has failed due to an initialization error.
20	TASK_ABORTED_ID	The task has been ended prematurely by request.
24	TASK_EXECUTION_DENIED_ID	The system has denied the execution of this task.
28	TASK_EXECUTION_DENIED_INSUFFICIENT_RESOURCES_ID	The system has denied the execution of this task because there are not enough resources available.

For convenience, the `CompletionCode` object has an ID value represented by an integer, which is available by calling the `CompletionCode.getId()` method. There are also corresponding ID values defined as constants in the `CompletionCode` class to enable the use of Java switch statements to easily perform logic based on particular `CompletionCode` values.

Below is a basic template for this type of Java switch statement:

```
CompletionCode cc = jobRecord.getCompletionCode();
int ccId = cc.getId();
switch (ccId) {
case CompletionCode.TASK_SUCCESSFUL_ID:
// Do something; (int value = 0)
break;
case CompletionCode.TASK_SUCCESSFUL_WITH_WARNINGS_ID:
// Do something; (int value = 4)
break;
case CompletionCode.TASK_FAILED_WITH_ERRORS_ID:
// Do something; (int value = 8)
break;
case CompletionCode.TASK_FAILED_WITH_RUNTIME_EXCEPTIONS_ID:
// Do something; (int value = 12)
break;
case CompletionCode.TASK_FAILED_ON_INITIALIZATION_ID:
// Do something; (int value = 16)
break;
case CompletionCode.TASK_ABORTED_ID:
// Do something; (int value = 20)
break;
case CompletionCode.TASK_EXECUTION_DENIED_ID:
// Do something; (int value = 24)
break;
case CompletionCode.TASK_EXECUTION_DENIED_INSUFFICIENT_RESOURCES_ID:
// Do something; (int value = 28)
break;
default:
// Unexpected
break;
}
```

In addition to accessing and reviewing a job's completion code, another useful feature of the `JobRecord` object is that it collects warning messages, error messages, and exceptions that have been logged during the job's execution cycle.

The following code sample illustrates the access of these properties from the `JobRecord` object:

```
JobLogMessage[] warningMessages = jobRecord.getWarningMessages();
JobLogMessage[] errorMessages = jobRecord.getErrorMesssages();
Throwable[] exceptions = jobRecord.getExceptions();
```

The `JobLogMessage` class is found in the `com.xenos.d2e.log` package. For information regarding the logging mechanism in , see "Message Handling" on page 25, and consult *OpenText Embedded Output Transformation Engine Javadoc*.

Chapter 5

Output Transformation Engine Exceptions

Output Transformation Engine exception handling involves the handling of various types of exceptions that the developer may encounter, all of which are found in the `com.xenos.d2e.exceptions` package. They are:

5.1 XenosException

The `XenosException` class is the base class from which most of the other Output Transformation Engine exceptions are extended. There are three constructors for XenosExceptions:

- `public XenosException(Exception cause, String plainTextMessage)`
- `public XenosException(Exception cause, Message message)`
- `public XenosException(Message message)`

The Message in the arguments is a `com.xenos.d2e.mail.messages` Message object that is used internally to relay informational messages between components. In custom programming for the Output Transformation Engine API, the developer should only require the use of the first constructor.

5.2 XenosIOException

The `XenosIOException` is thrown typically when using the `com.xenos.d2e.io` package for such instances as trying to create a `D2eInputStream`, in the `IoUtility` class.

5.3 XenosRuntimeException

The `XenosRuntimeException` extends `java.lang.RuntimeException` and behaves in a similar fashion. It may be thrown from within a method that does not explicitly declare it in a **throws** declaration.

5.4 SystemException

The `SystemException` is an exception issued by the system to indicate a failure, typically during system startup, and must be caught when calling `com.xenos.d2e.D2eProcess.startSystem()`.

5.5 JobInitException

The `JobInitException` is thrown by a job if a validation error or exception occurs when initializing the job before it is run.

5.6 UserNotifiableException

The `UserNotifiableException` is used by classes that want the exception to contain a message where the message can be localized and placed into a report or displayed to a user.

5.6.1 ExceptionUtil

The `com.xenos.d2e.exceptions` package also contains an `ExceptionUtil` class which provides some helpful methods for extracting information from a `XenosException`. These methods are as follows:

- `public static String getFullString(Throwable t)`

Gets a string that contains the details of the exception and all sub-exceptions.

- `public static String getStackTrace(Exception e)`

Gets a string that contains the stack trace of the exception.

- `public static String getLocalizedMessage(Exception e)`

Gets a localized message from an exception. If a pure localized method cannot be retrieved, a normal message or the name of the exception class is returned.

5.6.1.1 Example

```
 import com.xenos.d2e.exceptions.*;

...
try {
/*
* Start the engine
*/
m_d2eProcess.startSystem();
}
catch (SystemException se) {
String myDetails = ExceptionUtil.getFullString(se);
String myStackTrace = ExceptionUtil.getStackTrace(se);
String myRootCause = ExceptionUtil.getRootExceptionType(se);
String myLocalizedMessage = ExceptionUtil.getLocalizedMessage(se);

System.out.println(se.getDetails());
```

```
System.exit(se.getExitCode());
}
```

Chapter 6

Message Handling

Programs that interface with Output Transformation Engine through API calls will often require receiving (and possibly reacting to) messages that are issued during runtime.

The Output Transformation Engine architecture provides this capability through its own logging mechanism. This mechanism consists of a set of classes with which the user can interface to receive any messages that they are interested in.

Messages are broken up into two separate types:

- **System log messages**. System log messages are messages that are issued by system components (such as JobManager or LicenseAuthenticator).
- **Job log messages**. Job log messages are messages that are issued by job components (such as parsers and generators).

6.1 Setting Up Log Subscribers

To receive system log messages, you must create an object that implements the `com.xenos.d2e.log.ISystemLogSubscriber` interface.

To receive job log messages, you must create an object that implements the `com.xenos.d2e.log.IJobLogSubscriber` interface.

Note: Please refer to *OpenText Embedded Output Transformation Engine Javadoc* for these interfaces for specific information on what is required to implement them.

Creating an object that implements the desired interface(s) is only the first step to receiving messages. The Output Transformation Engine system must have access to the object(s) before any messages can be received.

To receive system log messages, users should provide a reference to their `ISystemLogSubscriber` object(s) to `D2eProcess` using the method `D2eProcess.addSystemLogSuscriber()`. This method can be called at any time, either before or after the system is started, but any messages that a system component issues before `addSystemLogSubscriber()` is called will not be sent to your `ISystemLogSubscriber`.

To receive job log messages, users should provide a reference to their `IJobLogSubscriber` object(s) to a JobRecord object by calling `JobRecord.addJobLogSubscriber()`. As with System log subscribers, this method can be called before or after a job starts, but it is recommended that this method be called before the `JobRecord` is submitted to `JobManager` for execution.

6.2 The LogSubscriberProfile Object

Both the `D2eProcess.addSystemLogSubscriber()` and `JobRecord.addJobLogSubscriber()` methods accept two parameters as input. The first is the appropriate subscriber instance (an instance of `ISystemLogSubscriber` or `IJobLogSusbscriber`) and the second is an instance of `LogSubscriberProfile`.

The purpose of the `LogSubscriberProfile` object is to specify the rules that determine what messages should be sent to the given subscriber. In other words, it acts as a filter.

Using the `LogSubscriberProfile` object, the API user has the ability to filter messages based on the following properties:

- **Component type name of the component that issued the message.** For example, the user could decide that they are only interested in messages that are issued by the JobManager and the AfpParser. For more information, see "Component Type Names" on page 27.
- **Instance name of the component that issued the message.** Certain components, especially job components, can have instance names because there can be more than one of those types of components at any given time and the instance name helps identify it. For example, the user may have an application defined that has two PdfGenerators running, one called `PdfGen1` and another called `PdfGen2`. If they only want to receive messages from `PdfGen2` they can use the instance name to filter messages from only that instance.
- **Type of message.** There are several types of messages, including Error messages, Warning messages, Information messages and Status messages. The user has the ability to receive only the message types that they are interested in.

Consult *OpenText Embedded Output Transformation Engine Javadoc* for this class for specific information on configuring the `LogSubscriberProfile`.

One benefit of having the rules defined separately from the subscriber object is that the same `LogSubscriberProfile` object can be used for multiple subscribers. In a dynamic web environment, for example, a static `LogSubscriberProfile` can be set up for all requests and reused.

Note: For simplicity of coding, the user has the option of setting a flag in the `LogSubscriberProfile` that allows a subscriber to receive all messages that are issued by the system and/or job. If this flag is set, the subscriber's code can simply ignore any messages that they are not interested in, rather than being required to subscribe to all messages that they want.

6.3 Component Type Names

As outlined in the previous section, messages can be filtered based on the type name of the component that issued the message. However, because the application's flexible architecture consists of “plug and play” components that are assembled at runtime, there is no central registry that contains all of the valid type names.

Here is the current list of component type names for the system components that issue messages:

- System.D2eProcess
- System.JobManager
- System.LicenseAuthenticator
- System.XdcApplicationResourceLoader

The project (d2eproj) file contains the name and type for each component. The following fragment of a d2eproj file demonstrates this:

```
<component type="afpparser" name="parser1">
```

where:

- **afpparser** is the generic component type for all AFP parsers.
- **name** is the specific instance of this particular component.

In addition to job components defined in the project file, there are two other job type components that are unique to each job. They are defined as:

- **Job**. Messages issued by the job itself.
- **Job.Mfct.** Master Font and Color Table - Font/Color correlation messages issued during the processing of the job.

Note: The `Job` and `Job.Mfct` components do not have instance names.

6.4 LogMessage Subclasses: JobLogMessage and SystemLogMessage

Every message that is sent to a subscriber is a subclass of `LogMessage`. These objects are pretty straightforward and the *OpenText Embedded Output Transformation Engine Javadoc* for these classes provides a good overview of their functionality.

`IJobLogSubscribers` receive JobLogMessage objects, which include the job name and a reference to the JobRecord object.

The following is an example of subscribing to receive all job level messages from all components:

```
// Subscribe to Errors that are issued by AfpParser's
 profile.subscribeToComponentTypeMessages("AfpParser",
 JobLogMessage.ERROR_MESSAGE);
 // Subscribe to System messages
 d2eProcess.addSystemLogSubscriber(this, profile);
 // Start the system
 d2eProcess.startSystem();
 ...
```

Chapter 7

Getting Started with Parameters

Parameters are configuration details that may be set in your `d2eSys` file or `d2eProj` file. They are usually used for providing values for variables within a component.

7.1 Creating Parameter Objects

To use parameters in your own components, you must use the `ParameterParm` class found in the `com.xenos.d2e.system.parm` package. You can set **name** and **value** pairs using this class.

To implement the `ParameterParm` object in your components, you must add the following type of line in your d2eproj file. The following example is taken from the `SampleIoHandlerFactory` project file.

Using a `ParameterParm` in your IOHandler, insert this into your d2eproj file.

```
 <iohandlerfactory instancename="WEBHANDLERFACTORY"
classname="com.xenos.d2e.examples.iohandler.SampleIoHandlerFactory">
 <parameter name="stream type" value="FILE"/>
 </iohandlerfactory>
```

Then utilize the `ParameterParm` to extract the data in your IOHandler.

The following is the corresponding code from the `SampleIoHandlerFactory` that retrieves the `stream type` parameter from the project.

```
 private final String TYPE_PARM = "stream type";
 ...
 // Retrieve our Parameter object from the m_factoryParm object that is loaded
 // in the init method.
 ParameterParm typeParm = m_factoryParm.getParameter(TYPE_PARM);
 type = typeParm.getValue();
 if(type.equalTo(FILE) == 0) {
 // FileInputStream - write to file
 is = (InputStream)new FileInputStream(location);
 }
```

The type variable will be set to `FILE`. You can have multiple parameter tags in your project file inside the same tag, however they need to all have unique name attributes.

The OpenText Professional Services team can create more complex parameters as a custom services project if required.

Chapter 8

Customizing I/O Access with IOHandlers

IOHandlers are used to override the engine's default way of performing I/O. The engine will read from and write to disk by default.

Examples of sources and destinations include:

- Byte arrays
- Databases
- Compressed/encrypted files
- Streamed resources
- Internet-based streams
- Document archival systems
- Stacked output

Instances of IOHandler factories may be defined in either the system configuration or application file. An instance definition consists of a unique name, the full class name of the factory, and parameters (if any). Each instance is initialized with its parameters, which are made fully available to the IOHandler Factory for use by custom code.

8.1 Developing UserIoHandlerFactories

To create a new `UserIoHandlerFactory`, you need to create a class that extends `com.xenos.d2e.io.UserIoHandlerFactory` and imports the `com.xenos.d2e.types` and `com.xenos.d2e.system.parm` packages. To your new class you need to add declarations for the following abstract methods:

```
public InputStream getInputDataHandler(JobRecord jobRecord, String location)
public D2EFilterInputStream getInputDataFilter(JobRecord jobRecord, FileDef fd)
public D2EFilterInputStream getInputRecordHandler(JobRecord jobRecord, FileDef fd)
public D2EFilterInputStream getInputRecordFilter(JobRecord jobRecord, FileDef fd)
public OutputStream getOutputDataHandler(JobRecord jobRecord, String location)
public D2EFilterOutputStream getOutputDataFilter(JobRecord jobRecord, FileDef fd)
public D2EFilterOutputStream getOutputRecordHandler(JobRecord jobRecord, FileDef fd)
public D2EFilterOutputStream getOutputRecordFilter(JobRecord jobRecord, FileDef fd)
public void release() throws IOException
```

You also need to set boolean member variables to indicate which types of handlers/filters have been implemented in this I/O Handler. For example, you could place these in your class constructor:

```
m_hasInputDataHandler = true;
m_hasInputDataFilter = false;
m_hasInputRecordHandler = false;
m_hasInputRecordFilter = false;
m_hasOutputDataHandler = true;
m_hasOutputDataFilter = false;
```

```
m_hasOutputRecordHandler = false;
m_hasOutputRecordFilter = false;
```

These flags default to **false** in the super class, so you only need to set those you want to use to **true**.

Now add code to the body of the method(s) that you will be using. The rest may simply return null, or in the case of `release()`, no method body is required as there is no return type. Each defined method must return the appropriate type of stream and may also throw an IOException. If the exception occurs in your code, you may either catch it and re-throw it or leave it uncaught so that it will be caught by the calling method.

8.1.1 Examples

```
public OutputStream getOutputDataHandler(JobRecord jobRecord, String location) throws
IOException {
// if calling this constructor throws an exception, IOUtility will catch it
FileOutputStream fos = new FileOutputStream(location);
return fos;
}
```

To test the class, define an instance in the system configuration or project, and reference the factory instance in a file definition using the instance name.

Example I/O Handler factory definition:

```
<!-- instanceName is the "logical" name for this instance -->
<!-- className is the Java class name and can include the package name -->
<iohandlerfactory instanceName="FACTORY1" className="SampleHandlerFactory"/ >
```

Example file definition which references the factory instance that was defined in the example above:

```
<filedeflist name="Fdoutput">
<filedef directives="{NOCRLF}" location="{d2eOutputPath}-{0}.pdf"
ioHandlerFactoryInstance="FACTORY1" cache="job"/>
</filedeflist>
```

For applications where input/output streams will be passed in by an external source, the stream(s) must first be passed in to the `JobRecord` after it has been constructed, then additional code is required in the `UserIoHandlerFactory` to retrieve the stream from within the appropriate method.

Example of setting the `InputStream` after the `JobRecord` has been created:

```
...
FileInputStream fis = new FileInputStream(fileName);
JobRecord.setInputStream(fis);
...
```

Example of retrieving the `InputStream` for use by the I/O Handler factory:

```
...
public InputStream getInputDataHandler(JobRecord jobRecord, String location) {
// calling the getInputStream() or getOutputStream() methods on the
// JobRecord will return the stream that was passed in when the
// JobRecord was first created.
```

```
 return jobRecord.getInputStream();
 }
 ...
```

If the I/O Handler Factory needs to make use of parameters, import the `com.xenos.d2e.system.parm` package, and then retrieve and make use of parameters using the `getParameter()` and `getValue()` methods. The variable `m_factoryParm` is declared as protected in `UserIoHandlerFactory`, and is available to classes that extend it. Parameters are name/value pairs and are therefore very flexible and simple to define.

Example of an I/O Handler Factory definition with parameters:

```
<!-- parameters need a name/value pair and are used by custom UserIoHandlerFactories -->
<iohandlerfactory instanceName="FACTORY1" className="SampleHandlerFactory">
<parameter name="stream type" value="file"/>
<parameter name="connection string" value="username:password"/>
</iohandlerfactory>
```

Example code that could be placed in any of the method bodies of the I/O Handler Factory:

```
 ParameterParm typeParm = m_factoryParm.getParameter("stream type");
 ParameterParm connectionParm = m_factoryParm.getParameter("connection string");
 // The value of the variable "type" should be the string "file" after this call
 String type = typeParm.getValue();
 // The value of the variable "connection" should be the string "username:password"
after // this call
 String connection = connectionParm.getValue();
```

8.2 Developing Stacked I/O Handler Factories

Output Transformation Designer uses Java's ServiceLoader facility at runtime to retrieve the list of registered I/O handler factories. Comprehensive information about the ServiceLoader class can be found in the Oracle Javadoc, which can be retrieved from:

http://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html

While loading the listing of I/O handler factories, Output Transformation Designer fetches a list of services that extend the `com.xenos.d2e.io.AbstractIoHandlerFactory` base class. For reference, the `Xenos-d2eVision-engine.jar` file, located in the `<install_home>\lib\common` folder, contains a `META-INF/services/com.xenos.d2e.io.AbstractIoHandlerFactory` file that registers the `com.xenos.d2e.xdc.component.indexer.StackedOutputHandlerFactory` implementation.

When a registered stacked I/O handler factory is loaded in Output Transformation Designer, the I/O handler is queried for a detailed list of parameters by calling the get `PropertyInfo()` method within the instance of a factory in order to provide parameter validation of the runtime parameters specified by the user at design time. API users can opt to override this method; consult the `com.xenos.d2e.parmdef.parm.PropertyInfo` class in the *OpenText Embedded Output Transformation Engine Javadoc* for more information.

Note: If the `com.xenos.d2e.parmdef.parm.PropertyInfo getFactoryParmInfo()` method is enabled and returns a null value, Output Transformation Designer presents the basic name/value pair parameters.

Chapter 9

Resource Version Control

Output Transformation Engine supports **Resource Version Control** (RVC) and user-written RVC components for AFP transformations. The purpose of the RVC is to allow users to store multiple versions of a resource in their archive system, such as a company logo in an AFP overlay or signature in a font, and still be able to retrieve the correct version of the resource from the cache when processing an archived AFP document.

There are two separate processes common to all AFP-based archival systems that utilize the RVC technology: the load process and the retrieval process.

The **load process** typically involves loading large AFP files which contain hundreds or thousands of individual documents and all associated resources into the archive system. This step uses an AFP2AFP transform project, which includes the Indexer and Index Writer components configured to insert document breaks for each individual document based on user-specified criteria.

As resources are processed during the AFP2AFP load process, a Resource Version Controller class will compare each resource with the previously archived resources to determine whether the resource already exists in the archive, or whether it needs to be added to the archive. This controller is custom-implemented to meet specific business requirements and to interface with the archival database in the target environment.

Each resource is assigned a unique key when it is added to the archive by the controller. The controller will typically use binary checksums to compare resources and to index the resource keys.

Output Transformation Engine will store these resource keys in special NOP comment records in the generated AFP documents. A single NOP will appear at the beginning of each document to specify version information for document level resources, such as form definitions, and each page will contain an NOP listing the resource keys for all resources used on that page.

The **retrieval process** occurs when the AFP documents are retrieved from the archive and commonly converted to PDF, TIFF, or PNG for on-demand presentment to end users. Output Transformation Engine will parse the NOP record during this transformation, so that the appropriate version of each resource is used. In combination with the Resource Cache, a single version of a resource will be processed once and only once for all output files. This results in faster retrieval times and lower memory requirements to archive AFP resources.

9.1 Specifying the RVC to Use

You can specify the RVC configuration at the system-wide level in the system configuration (d2esys) file, or at the project level in the d2eproj file.

The following is an example of system level configuration:

```
<resourcemanager resourceversioncontroller="com.xenos.d2e.resource.SampleRvc">
<RvcAttributes>
<parameter name="logfile" value="c:\path\to\log\SampleRvc.log"/>
<parameter name="resourcePath" value="c:\path\to\resources\"/>
</RvcAttributes>
</resourcemanager>
```

When using the project file setup, the configuration is similar:

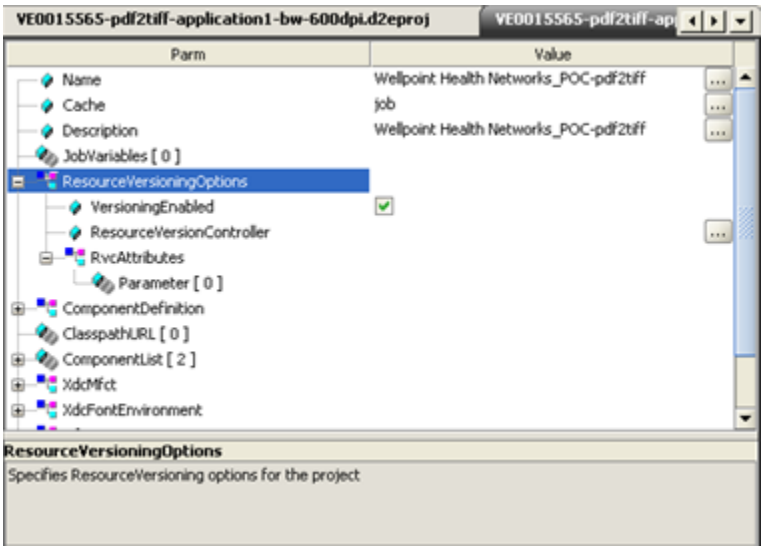


Figure 9-1: ResourceVersioningOptions configuration parameters

There is one key difference in the project level configuration: the presence of the **VersioningEnabled** parameter. This option may be useful in environments that use the system level configuration and have many different project types running under the same engine. In those instances, it may be required to disable the RVC system for certain projects, in which case this parameter should be set to false.

In both cases, the `resourceversioncontroller` attribute specifies the fully qualified class name of the RVC to use. This class must implement the `com.xenos.d2e.resource.IResourceVersionController` interface. Refer to *OpenText Embedded Output Transformation Engine Javadoc* and SampleRVC source code provided.

Any number of name/value parameters can be passed into the RVC. These can be used to configure database connection parameters or other contextual properties. The property names to use depends upon the underlying implementation of the RVC. Generally, the property names are case-sensitive, thus care should be taken to

ensure the names are entered properly. If you cut and paste the property names from the RVC's documentation, be sure not to accidentally paste in a space character.

Chapter 10

Indexer and Index Writers

The objective of this section is to explain:

- How to use and integrate the Indexer component and the Index Writer
- How to develop new custom Index Writers
- How to use the IXifIndexFilter (exposed in the Indexer component) to modify the values that the Indexer gets.

10.1 Indexer/Index Writer Functional Summary

The Indexer component includes the Indexer and several Index Writers with the capability for other Index Writers to be written using the Output Transformation Engine API.

10.1.1 Indexer

The function of the Output Transformation Engine Indexer is to gather information from the following sources:

- XFT Fields
- Page Groups/TLEs (gathered from AFP files)
- XIF Comments (gathered, for example, from AFP NOPs, Metacode Comment records, or TIFF tags if the ASCIITagsAsComments parameter is set)

10.1.2 Index Writer

The Index Writers write out the contents of the Indexer in various formats. Three generic Index Writers are provided:

- An IBM CMOD Writer which writes out a file in IBM CMOD format.
- An XML Writer which writes out the index as an XML file using a fixed format, as an example of how OpenText Professional Services or customer developers could write their own.
- A FileNet Index Writer, which functions as a CSV index writer.

10.2 Overview of Indexer/Index Writer Packages

This section describes packages related to creating indexes.

10.2.1 com.xenos.d2e.transform.common.xif.index

This package contains three interfaces and one class. These are all related to the creation of Indexes and define how the indexing classes interact with each other.

IXifIndexFilter

This interface defines a class that can inspect the data mined in order to create an index. A class that implements this interface is responsible for adding index values to the XifIndexDocument. The implementing class can also modify the values that have been mined in order to change field names or format the data used for indexing.

XifDocumentIndex

This class is what stores the index information for each logical document. Each XifDocument contains one of these objects.

XifIndexComponent

The Indexer component implements this.

IXifIndexWriter

The BaseIndexWriter implements this.

XftIndexTable

This class contains a representation of the data in an XftTable.

XifIndexEntry

This class is used to store different types of index information in the XifDocumentIndex.

10.2.2 com.xenos.d2e.xdc.component.indexer

This package contains the Indexer component and its parameters.

Indexer

This class is responsible for mining the indexing information from the events being passed to it. It can extract data from XFT, AFP Page Groups/TLEs and Comment records. It may also perform a callback to an IXifIndexFilter and allow it to add the data to the index.

SampleIndexFilter

This class is an example of an index filter that filters XFT events for the `sample-afp2-pdf.d2eproj` project and reformats the **Policy_Number** and **Letter_Date** fields before adding them to the index.

StackedFileOutputStream

Performs writing to a stacked output file. Used in conjunction with the StackedOutputHandlerFactory.

StackedOutputHandlerFactory

Responsible for creating StackedFileOutputStreams when `getOutputDataHandler()` is called.

10.2.3 com.xenos.d2e.xdc.component.indexwriter

This package contains the IndexWriter component, its parameters and the base class for creating index writers.

BaseIndexWriter

This class is an abstract class that defines which methods need to be implemented to create an index writer.

IndexWriter

This class is responsible for creating the IndexWriter that has been set in its parameters.

OnDemandIndexWriter

Writes out an index in the OnDemand format.

XmlIndexWriter

Writes out the index as an XML file.

10.3 Writing an Index Filter

To write an index filter:

1. Create a class that implements the **IXifIndexFilter** interface.

   To do this, you must import the following:

   ```
   import com.xenos.d2e.transform.common.xif.index.*;
   ```

2. Define your class:

   ```
   public class SampleIndexFilter  implements IXifIndexFilter
   ```

   Your Index Filter can accept parameters that have been defined for it in the **FilterProperties** section of the project. These parameters are passed in as a **ParmHandler** named parameter as an argument to the **init** method that you must implement.

   To use these parameters, do the following:

   a. Declare a class member variable. For example:

      ```
      private FilterPropertiesParm m_parms;
      ```

   b. Cast the ParmHandler to this variable in your **init** method:

      ```
      m_parms = (FilterPropertiesParm)parms;
      ```

3. Define the **init** method. The init method is defined as follows:

   ```
   public void init(IXifIndexComponent  component, ParmHandler parms)
   ```

   The **FilterPropertiesParm** contains a collection of ParameterParm objects which are simply name/value pairs.

   To access these objects, you can iterate through them retrieving each one by its index. For example:

   ```
   HashMap m_props = new HashMap();
   int count = m_parms.getPropertiesCount();
   for (int i = 0; i < count; i++) {
      ParameterParm pp = m_parms.getProperties(i);
      m_props.put(pp.getName(), pp.getValue());
   }
   ```

   This code is simply retrieving each of the **ParameterParm** objects from the **FilterPropertiesParm** and storing their name and value in a HashMap with the ParameterParm name as the key.

4. Implement the **startNewIndex** method.

   This method is called whenever a new XifDocument is created. You can add a method body to your method if you are interested in new documents.

   ```
   public void startNewIndex(XifDocumentIndex  docIndex)
   ```

5. Implement the **filterComment** method.

This method is called whenever a comment is found in the print stream. This will only be called if the filterComments parameter is set to Enabled in the Indexer parameters.

```
public void filterComment(int  docPageNumber, XifDocumentIndex index, String
attribValue)
```

6. Implement the **filterTle** method.

   This method is called whenever a TLE is found in the print stream. This will only be called if the **filterTle** parameter is set to Enabled in the Indexer parameters.

```
public void filterTle(int docPageNumber,  XifDocumentIndex docIndex, String
pageGroupName, String attribName, String  attribValue)
```

   The **filterXft** method is probably the most useful in the class. It is called for every XftField processed on the XifPage. It will only be called if the **filterXft** parameter is set to Enabled in the Indexer parameters.

```
public void filterXft(int docPageNumber,  XifDocumentIndex docIndex, String
xftFieldName, String xftFieldValue)
```

   The **xftFieldName** is the name of the field and the **xftFieldValue** is the value of the field. An example of their usage is as follows:

```
String name = xftFieldName.toUpperCase();
String value = xftFieldValue;
if (xftFieldName.equalsIgnoreCase("Policy_Number"))  {
value = reformatPolicyNumber(xftFieldValue);
} else if (xftFieldName.equalsIgnoreCase("Letter_Date"))  {
value = reformatDate(xftFieldValue);
}
docIndex.addIndex(name, value);
```

   The line **docIndex.addIndex** actually adds the name and value to the **XifDocumentIndex**, thereby ensuring it gets passed to the IndexWriter.

7. Implement the **filterXftTableRow** method.

   This is called for every row event received from XFT. The data from the row is actually contained in the rowResult.

```
public void filterXftTableRow(int  docPageNumber, XifDocumentIndex docIndex,
String xftFieldName, XftTableRowResult  rowResult)
```

   The **finalizeIndex** method is called prior to the index file being written out. This method allows the user to modify the index values just before sending them to the Index Writer.

```
public void finalizeIndex(XifDocumentIndex  docIndex)
```

8. Implement the **terminate** method.

   This method is used for any cleanup activity that needs to be performed. It is called when the job exits. Keep in mind that when the component shuts down, there still may be more calls to finalizeIndex().

```
public void terminate(IXifIndexComponent  component)
```

10.4 Writing an Index Writer

To write an index writer:

1. Create a class which implements the **BaseIndexWriter** interface.

   To do this you must import:

   ```
   import com.xenos.d2e.xdc.component.indexwriter;*
   ```

2. Define your class:

   ```
   public class CustomIndexWriter  extends BaseIndexWriter
   ```

   Your Index Writer can accept parameters that have been defined for it in the **CustomWriterSetting** section of the project. These parameters are passed to the init method as a **ParmHandler** named *<parms>;* you must implement the init method.

   To use these parameters, do the following:

   1. Declare a class member variable. For example:

      ```
      private WriterSettingsParm m_parms;
      ```

   2. Cast the ParmHandler to this variable in your init method:

      ```
      m_parms = (WriterSettingsParm)parms;
      ```

3. The init method is defined as follows:

   ```
   public void init(ParmHandler  parms, IoUtility ioUtil, int writerNumber)
   ```

   The **WriterSettingsParm** contains a collection of IndexWriterNameValuePairParm objects which are simply name/value pairs. To access these objects, you can retrieve the value by name or iterate through them retrieving each one by its index. For example:

   ```
   IndexWriterNameValuePairParm  pp = m_parm.getWriterSettings(OUTPUT_KEY);
   or
   Enumeration keys = m_parm.getWriterSettingsKeys();
   while (keys.hasMoreElements())  {
       IndexWriterNameValuePairParm  nameValue =
  m_parm.getWriterSettings((String)keys.nextElement());
   }
   ```

   This code is simply retrieving each of the **IndexWriterNameValuePairParm** objects from the **WriterSettingsParm** and storing their name and value in a HashMap with the IndexWriterNameValuePairParm name as the key.

   The second argument to the init method is the IOUtility. The IOUtility provides factory methods for creating objects to read and write data from sources and destinations described by Locator objects. The data format is specified by Directive objects, which are constructed by the factory methods using the I/O directive prefixes. This IOUtility is to be used if reading and writing the index to a file is required or if an IoHandler object is required.

The following package must be imported to make use of the IOUtility and D2eInputStream and D2eOutputStream:

```
import com.xenos.d2e.io.*;
```

Please see the Javadocs for more information about the **comn.xenos.d2e.io.IoUtility** and its usage. This class is found in the framework.jar.

4. Implement the **finalizeIndex** method.

   The **finalizeIndex** method performs any finalization on the index file that may need to occur. This method is called from the generator when it has begun shutting down after it has received an end of file event representing the end of the input file. This method may be responsible for writing a footer entry, closing a file, or closing a database connection, for example.

```
public abstract void finalizeIndex()  throws XenosException;
```

   This method may throw the **XenosException** so you will need to import the following:

```
import com.xenos.d2e.exceptions.*;
```

- Implement the **processIndex** method.

The generator calls this method when a document is closed. It is responsible for processing the index information for one logical document. The props argument that it accepts is a **java.util.ArrayList** of java.util.ArrayList objects. Each entry in the ArrayList contains a **XifIndexEntry** object.

```
public void processIndex(ArrayList  props) throws XenosException
```

Each **ArrayList** object in the props ArrayList contains the index information for one complete index entry for the document. Keep in mind that one document is allowed to have multiple index entries containing the same fields. This may be useful for documents that need to be indexed on multiple things. For example, one document may contain a list of account numbers that you wish to index on.

The **XifIndexEntry** contains a type property that can be accessed through **getType()**. The type can be one of five:

- `public static final int XFT_INDEX_FIELD = 1;`
- `public static final int XFT_INDEX_TABLE = 2;`
- `public static final int INDEX_TLE       = 3;`
- `public static final int INDEX_COMMENT   = 4;`
- `public static final int INDEX_INFO   = 5;`

The **XFT_INDEX_FIELD** represents an XFT field and is stored in an XifIndexNameValuePair.

The **XFT_INDEX_TABLE** represents an XFT table field and is stored in an XftIndexTable.

The **INDEX_TLE** represents a TLE entry and is stored in an XifIndexNameValuePair.

The **INDEX_COMMENT** represents a Comment record and is stored in an XifIndexNameValuePair.

The **INDEX_INFO** represents information from the engine and is stored in an XifIndexNameValuePair. It can have one of the following as its name:

This property is the byte offset where this document starts in the output.

```
XifDocumentIndex.DOCUMENT_SIZE_KEY
```

This value is the size of this document in bytes.

```
XifDocumentIndex.DOCUMENT_NUMBER_KEY
```

This value is the number of this logical document in the input file.

```
XifDocumentIndex.DOCUMENT_OUTPUT_KEY
```

This value is the name of the output file that this document was written to.

The **XftIndexTable** contains the table name and rows.

The rows are stored as an ArrayList of strings that represent the cell value.

You can get the table name as defined in XFT by:

```
public String getTableName()
```

You can also get the rows using various methods. Since XftIndexTable extends Hashtable, you have the option to get the keys which are **XftIndexTable.TableKey** objects or by:

```
public ArrayList getRow(int  row)
```

which will get you the row indexed by its row number. You can iterate through all the rows in order by:

```
int numRows = table.getNumRows();
for (int j = 0; j < numRows;  j++) {
ArrayList row = table.getRow(j);
for (int col = 0; col < row.size();  col++) {
    String  colValue = (String) row.get(col);
}
}
```

If you use the Hashtable method, you are not guaranteed to get the rows in order.

```
Enumeration e = table.keys();
while(e.hasMoreElements()) {
XftIndexTable.TableKey key =  (XftIndexTable.TableKey)e.nextElement();
System.out.print("Row:  " + key.getRow() + " ");
ArrayList tmp = (ArrayList)table.get(key);
for (colCount = 0; colCount  < tmp.size(); colCount++) {
```

```
    System.out.print((String)tmp.get(colCount)+  "\t");
}
System.out.println("");
}
```

where table is an XftIndexTable for both examples.

The **XftIndexTable.TableKey** stores the row number that you can access using:

```
XftIndexTable.TableKey key =  (XftIndexTable.TableKey)e.nextElement();
long keyval = key.getRow();
```

The **XifIndexNameValuePair** stores a name and a value. Its accessors are:

```
public String getName()
public String getValue()
```

and its constructor:

```
public XifIndexNameValuePair(String  name, String value)
```

10.5 Viewing Index Filter and Writer Sample Code

com.xenos.d2e.examples.indexfilter.SampleIndexFilter

This example shows how to write a simple Index Filter. It is only concerned with the XFT events. It simply looks for the `Policy_Number` field or the `Letter_Date` field and reformats it. It then inserts the reformatted values in the index.

com.xenos.d2e.examples.indexwriter.sampleIndexWriter

Writes out the XFT fields defined in the Indexer to an XML file.

Chapter 11

Replace Text Component

11.1 Objective

The objective of this section is to explain the following:

- How to use the Replace Text component.
- How to develop new Replace Field Validators.

11.2 Replace Text Functional Summary

The replace text component is used to change the text that appears in the output file and is captured by an XFT field.

To make this functionality versatile, we have included the ability to write field validators, using the Output Transformation Engine API, that are responsible for matching the value of an XFT field as well as performing the text replacement.

The function of the Replace Text component is to:

- Inspect the XFT field that has been defined for modification.
- Verify that the value of this XFT field is meant to be modified.
- Replace the XFT field value and replace the underlying data on the XifPage.

11.2.1 Overview of Replace Text Packages

11.2.1.1 com.xenos.d2e.xdc.component.replacetext.validators

This package contains the **IReplaceFieldValidator** interface along with the validators that are selectable from the Replace Text component parameters in the application.

11.2.1.2 IReplaceFieldValidator

This interface defines a class that can test if a given XFT Field value should be replaced or not. It also performs the actual test replacement.

11.2.2 Writing a Replace Text Field Validator

To write a Replace Text field validator:

- Create a class that implements the `IReplaceFieldValidator` interface. Your new class must import:

```
import com.xenos.d2e.xdc.component.replacetext.validators.IReplaceFieldValidator;
```

- Define your class:

```
public class SampleFieldValidator implements IReplaceFieldValidator
```

- The `IReplaceFieldValidator` interface requires you to implement the following methods:
  - `init`
  - `setJobVariables`
  - `isMatch`
  - `modifyField`

11.2.2.1 init Method

```
/**
* Initializes this Validator.
* @param parm The ValidatorParm object for this Validator
* @param fieldName The name of the field that this Validator is
* being used for.
*/
public void init(ValidatorParm parm, String fieldName);
```

Your field validator can accept parameters that have been defined for it in the `ValidatorParm` section of the project. These parameters are passed in as a ParmHandler named parm as an argument to the init method that you must implement. You must import the parm package:

```
import com.xenos.d2e.xdc.component.replacetext.parm.*;
```

To use these parameters, declare a class member variable. For example:

```
private ValidatorParm m_parms;
```

Then, cast the `ParmHandler` to this variable in your init method:

```
m_parms = (ValidatorParm)parms;
```

From the `ValidatorParm` object you can get the `searchText`, `replaceText` and `exactMatch` parameter values:

```
String searchText = m_parms.getSearchText();
String replaceText = m_parms.getReplaceText();
boolean exactMatch = m_parms.getExactMatch();
```

You can also get the `CustomValidatorParm` which contains the custom settings and the classname of the custom validator:

```
CustomValidatorParm c_parms = m_parms.getCustomValidator();
String className = c_parms.getClassName();
```

There are also custom settings you can use from the `CustomValidatorParm`. These settings are stored in a `VectorParameterParm` object which contains a collection of `ParameterParm` objects, which are simply name/value pairs. To use these objects, you must import the following package:

```
import com.xenos.d2e.system.parm.*;
```

To access these objects, you can iterate through them retrieving each one by its index. For example:

```
VectorParameterParm v_parms = m_parms.getCustomValidator().getCustomSettings();
HashMap m_props = new HashMap();
int count = v_parms.getParametersCount();
for (int i = 0; i < count; i++) {
ParameterParm pp = v_parms.getParameters(i);
m_props.put(pp.getName(), pp.getValue());
}
```

This code is simply retrieving each of the `ParameterParm` objects from the `VectorParameterParm` and storing their name and value in a HashMap with the `ParameterParm` name as the key.

11.2.2.2 setJobVariables Method

```
/**
* Sets the JobVariables  to be used for substitution
* @param jobVars The jobVaraiables.
*/
public void setJobVariables(Properties jobVars);
```

This method is called by the Replace Text component to give the field validator access to the job variables associated with this job. These job variables can be used by the `com.xenos.d2e.system.PropertyFormat` class to perform substitutions in a string. The `PropertyFormat.fixJobVariables` method will replace all occurrences of job variables in the string passed in with the appropriate job variable. For more information, see "Job Variables" on page 16.

```
PropertyFormat.fixJobVariables(m_parms.getSearchText(),  m_jobVariables)
```

This allows you to use job variables in your search or replace strings that can be set upon job submission or by the command line.

11.2.2.3 isMatch Method

```
/**
* Returns <code>true</code> if the value of the field passed in
* matches the SearchText parameter.
* @param value The String value of the Field that we wish to match
* against.
* @return <code>true</code> if the value of the field passed in
* matches the
* SearchText parameter. <code>false</code> otherwise.
*/
public boolean isMatch(String value);
```

The `isMatch` method is used if the ExactMatch parameter is set to **true**. This method is responsible for checking if the field value is appropriate for substitution. For example, you may make this method check if the field value has the appropriate format such as ####-####-XX or you could check if the field value equals the SearchText parameter. If this method returns true, then the `modifyField` method is called with the field value.

You may use the parameters that you were given in the init method to obtain your `searchText` value:

```
String searchText = m_parms.getSearchText();
```

Or, if the `SearchText` may have job variables in it:

```
    PropertyFormat.fixJobVariables(m_parms.getSearchText(),m_jobVariables)
```

Or you may use any of the custom parameters that were passed in as described for the `init` method. For more information, see “init Method” on page 50.

11.2.2.4 modifyField Method

```
/**
* Modifies the field value according to the parameters.
* @param value The String value of the Field that we wish to substitute.
* @return The input string that has been modified.
*/
public String modifyField(String value);
```

This method is responsible for performing the field value replacement. The string passed in is the value of the field that was extracted. The string that is returned is the modified field value. This method will always be called if the `ExactMatch` parameter is set to **false** or will be called if the `ExactMatch` parameter is set to **true** and the `isMatch` method returns true.

You may use the parameters that you were given in the init method to obtain your `replaceText` value:

```
String replaceString = m_parms.getReplaceText();
```

Or, if the SearchText may have job variables in it:

```
PropertyFormat.fixJobVariables(m_parms.getReplaceText(),m_jobVariables)
```

Or you may use any of the custom parameters that were passed in as described for the `init` method. For more information, see "init Method" on page 50.

Chapter 12

Custom Components

12.1 User Extensions with Custom Components

User Extensions to Output Transformation Engine enable developers to insert their business logic in the engine's workflow by using the API to build custom components. These components receive events from the component located before them in the engine's workflow. A component that sends events down the workflow stream is called a **producer**, and the component that receives events from a producer is called a **consumer**. A component can be both a producer and a consumer, both receiving events and producing them.

These custom components are sometimes referred to as **User-Written Extensions** if written by the customer, or as **d2e Services Extensions** when written by OpenText Professional Services consultants.

12.1.1 Creating a Custom Component

To create a new custom component, you must extend from the `com.xenos.d2e.xdc.common.BufferedBaseComponent` abstract class. Please see *OpenText Embedded Output Transformation Engine Javadoc* for more detailed method descriptions

This class has five abstract methods that must be implemented as follows:

```
public String getComponentClass()
```

The **getComponentClass** method returns the class name of the component without the package prefix.

For example:

```
 public String getComponentClass() {
 return "SampleUserComponent";
 }
public String getDescription()
```

The **getDescription** method returns a short text description of the component.

```
 public String getDesccription() {
 return "This is what my component does";
 }
public int getMaxConsumers()
```

The **getMaxConsumers** method returns the maximum number of consumers that this component will send events to. A value of 0 indicates that this component sends events to an unlimited number of consumers. The following example shows a component that only allows one consumer to be connected to it.

```
 public int getMaxConsumers() {
 return 1;
```

```
 }
public int getMaxProducers()
```

The **getMaxProducers** method returns the maximum number of producers that this component will receive events from. A value of 0 indicates that this component receives events from an unlimited number of Producers. The following example shows a component that only allows one producer to be connected to it.

```
 public int getMaxProducers() {
 return 1;
 }
public void processPageEvents(java.util.ArrayList a_events, IXdcProducer producer)
```

The **processPageEvents** method accepts an array list of all of the `XdcEvents` that belong to a page along with the `XdcProducer` that produced these events. The following example shows a component that passes all of its events to its consumers.

```
 /**
 * Process a collection of XdcEvents. And pass them
 * along to all of my consumers.
 * @param a_events The XdcEvents to process
 * @param producer The IXdcProducer who sent the event.
 */
 public void processPageEvents(ArrayList a_events, IXdcProducer producer) {

 // pass event to all the consumers
 IXdcConsumer consumer;
 Object[] consumers = m_consumers.toArray();

 Object[] events = a_events.toArray();

 for (int i = 0; i < events.length; i++) {
         System.out.println("event: " + i);
         XdcEvent event = (XdcEvent)events[i];

         if(dc.isEnabled()) {
          dc.log(Debug.HI,event.toString(),producer.toString());
         }

         sendEvent(event);

         // check for end of file
         if ( event == null ) {
             m_manager.componentFinished(this);
         }
         else {
       if (event.getXdcEventType() == XdcEvent.BINDER) {
                 XifBinder binder = (XifBinder)event;
                 if (binder.getBinderType() == XifBinder.PAGE) {
                     if (((XifPage)binder).isEndOfFile()) {
                        // let ComponentManager know we're finished
                       m_manager.componentFinished(this);

                     }
                 }
             }
         }
     }
 }
```

12.1.2 Configuring XDCEvents

There are two types of events that the component may receive:

- `XifBinder`. Contains document and page related events.
- `XftEvent`. Contains information generated from the XFT Field Technology component. This information is comprised of data that has been mined from a page.

12.1.2.1 XifBinder Classes

In practice, the `XifBinder` classes are the XDCEvent types that are most commonly used by custom components. They represent the content and structure of documents in the XIF architecture model.

The `XifBinder` event can be one of three types:

- `XifDocument`. Indicates a beginning of a logical document.
- `XifPage`. Contains data representing a page within a document.
- `XifEndOfDocument`. Indicates the end of a logical document.

12.1.2.2 XifElementUtil Class

When processing `XifPage` events in extension components, API developers have the ability to add, remove and modify `XifElement` objects associated with each page. The Javadocs for the classes with the `com.xenos.d2e.transform.common.xif` package provide details about this robust functionality.

In addition to having direct access to objects in the primary XIF package structure, the class `com.xenos.d2e.transform.api.xif.XifElementUtil` provides a set of convenience methods that offer solutions for some of the common requirements of API users.

12.1.2.2.1 Adding an Image to an XifPage

Custom components are commonly developed to enhance output documents by adding images to one or more pages. They are typically corporate logos, signatures and in the case of financial institutions, sometimes check images. The `XifElementUtil` provides several methods to facilitate this (each of which are described in the Javadoc). Below is a code sample that illustrates this concept. The full source code for this component (`com.xenos.d2e.examples.usercomponents.InsertImagesComponent`) can be found in the source code samples.

```
/**
 * A sample private helper method designed to be used  in a custom subclass
 * of com.xenos.d2e.xdc.common.BufferedBaseComponent
 * <p>
 * It demonstrates how to insert an image onto a XifPage  using the
 * com.xenos.d2e.api.xif.XifElementUtil utility class.
 *
 * @param xifPage The XifPage object that the image  should be added to.
 *
```

```
 * @param imageFileData A byte array containing the  entire contents of
 * the image file that is to be added to the given  XifPage. The
 * XifElementUtil class will inspect the bytes to  determine the
 * actually image format, which is any of the types  defined in the
 * com.xenos.d2e.api.xif.ImageTypeEnum class (such  as JPEG or TIFF).
 */
private void addImageToPage(XifPage xifPage, byte[] imageFileData)  throws Exception {
    int oneInch = 7200;
    // Add the same image to the page  three times, in three different
    // physical locations, and in three  difference sizes (scaling)

    // Image 1: Use the full size of  the image.
    XifElementUtil.addImageToPage(m_manager,  xifPage, imageFileData,
              1*oneInch,   // xPos (top left corner)
              2*oneInch,   // yPos (top left corner)
              0,           // width,  0  means use actual image width
              0,           // height, 0 means  use actual image height
              1);          // page # (always 1, unless  image data
                          //  is a multi-page TIFF file)
    // Image 2: Constrict & scale  to only 2 inches wide, auto-detect
    //          height  to maintain aspect ratio.
   XifElementUtil.addImageToPage(m_manager,  xifPage, imageFileData,
              1*oneInch,   // xPos (top left corner)
              5*oneInch,   // yPos (top left corner)
              2*oneInch,   // width (desired)
              0,           // height, 0 means  detect to keep aspect ratio
              1);          // page # (always 1, unless  image data
                          //  is a multi-page TIFF file)
    // Image 3: Now do a 3 inch x 3  inch square image and do not preserve
    //          the  aspect ratio
    XifElementUtil.addImageToPage(m_manager,  xifPage, imageFileData,
              1*oneInch,   // xPos (top left corner)
              7*oneInch,   // yPos (top left corner)
              3*oneInch,   // width, 0 means use actual image width
              3*oneInch,   // height, 0 means use actual image height
              1);          // page # (always 1, unless  image data
                          //  is a multi-page TIFF file)
}
```

12.1.2.2.2 Joining Images on an XifPage

Typically, XifElementUtil is used to add images (XifImageElements) to the page one at a time, but functionality can also be set to add multiple images to a page simultaneously. The addition of separate images is suitable for most circumstances, but some users may notice degradation in speed when a single page contains many images (especially when printing the page), in which case it is recommended you try combining the images on the page. To combine the XifImageElements, the CombineImageParameters class is used as the method input argument with CombineImageResults as the return type. Below is a code sample that illustrates this concept.

The following fields are for CombineImageParameters:

```
public XifPage page;
    /**
         *  The Collection of XifImageElement to be combined.
         */
    public Collection<XifImageElement>  imagesToCombine;
    // Final dpi for the combined XifImageElement
    public int xDpi = 72;
    public int yDpi = 72;
    /**
     * Whether or not replace  the original images
     * When true, the method will  remove the old images
     * from the page and add the  new combined image in
```

```
      * their place
      */
 public boolean replaceImages = true;
```

The following fields are for CombineImagesResults:

```
/**
         * The XifImageElement that was created
         */
    public final XifImageElement xifImageElement;
```

Before combining any XifImageElements, there are a few limitations when it comes to combining images that users should be aware of:

- While either black and white or color XifImageElements can be collected in the same set of images, you cannot mix the two individual types into a single image set.
- Overlapping of XifImageElements is not supported.
- Since combining multiple XifImageElements into one is generally a memory intensive feature, especially when there are spaces between the original XifImageElements, it is recommended the original XifImageElements be aligned with minimal spacing in between them.

12.1.2.2.3 Extracting Data from XFTEvents

There are several types of **XftEvents**. The following example illustrates some of the various XftEvent types and how to extract field values:

```
if (xftevent.getType() == XftEvent.XFT_EVENT_DOCUMENTSTART) {
    // cache xft doc name
    m_currentDocName = xftevent.getDocumentName();
    System.out.println("Document start: " + m_currentDocName);
}
else if (xftevent.getType() == XftEvent.XFT_EVENT_DOCUMENTEND) {
    // We have the end of a document
    // We could do some stuff here
    System.out.println("Document End");
}
else if (xftevent.getType() == XftEvent.XFT_EVENT_SECTIONSTART) {
    // We have the start of an Xft Section
    m_currentSectionName = xftevent.getSectionName();
    System.out.println("Starting Section: " + m_currentSectionName);
}
else if (xftevent.getType() == XftEvent.XFT_EVENT_SECTIONEND) {
   // are we in section change doc breaking mode?
    System.out.println("Section End");
}
else if (xftevent.getType() == XftEvent.XFT_EVENT_FIELDPROCESSED) {
    String fieldName = xftevent.getTextFieldName();
    String fieldValue = null;
    System.out.println("Field Processed: " + fieldName);
    // is this is the field we are interested in?
    // get field value (if we already haven't)
    if (fieldValue == null) {
        try {
            fieldValue = xftevent.getTextFieldResult().
            getResult().getResultString();
            if (fieldValue != "") {
                System.out.println("Field: " + fieldName + "=" + fieldValue);
            }
        }
        catch (Exception e) {
```

```
            e.printStackTrace();
            fieldValue = null;
        }
    }
}
```

12.1.2.2.4 Adding a Component to the components.xml File

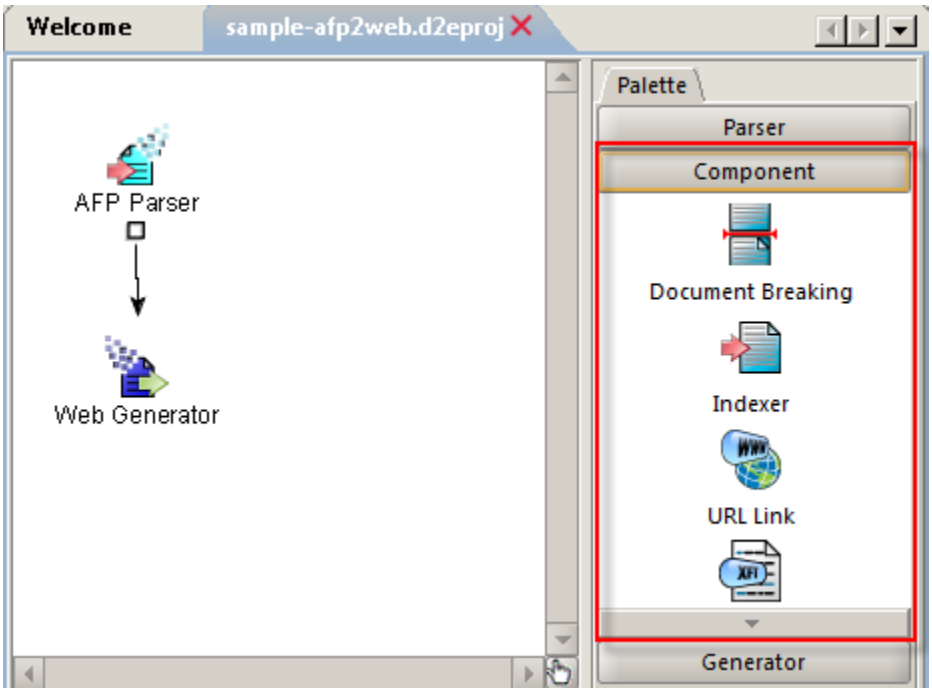
Custom components must be added to a `components.xml` file before they can be used. If you are deploying your first custom component, you need to create this file. If you already have custom components, the **components.xml** file will already exist. Add a new object section (from <object name="Component"> to </object>) to define the new component.

Include the following information in the `components.xml` file. Each custom component will contain a separate object section:

```
<?xml version="1.0" encoding="UTF-8"?>
<ComponentId>
<object name="Component">
<string name="Name" value="mycustomcomponent"/>
<string name="Part" value="VEXTC"/>
<string name="Description" value="This is my custom component."/>
<string name="ClassMain" value=" com.my.component.package.MyCustomComponent "/>
<string name="ClassParm" value=""/>
<string name="Package" value="com.my.component.package "/>
<string name="TabName" value="Component"/>
<string name="XftRequired" value="no"/>
<string name="ShowParameters" value="no"/>
</object>
</ComponentId>
```

Define the elements in the **Component object tag** as follows:

- **Name**. An all-lowercase version of the classname of your component (for example, **mycustomcomponent**).
- **Part**. This value must always be `VEXTC` which is a **V**ision **Ext**ension **C**omponent..
- **Description**. A short description of the component for Output Transformation Designer to display. This is the readable component name (for example, **My Custom Component**).
- **ClassName**. The fully qualified classname of the Component class.
- **ClassParm**. The complete path to your Parameter class (if you have one).
- **TabName**. The name of the tab in the Palette in which this component shows up. To add this component to the standard Output Transformation Engine components list, specify **Component**. To add it to a custom tab, enter a name for the custom tab. If the tab does not currently exist, it will be created. The sort order will always be alphabetical.



Note: Do not enter **Parser** or **Generator**.

- **XftRequired.** Yes or No. If this component requires the XFT component to be present in the project to function properly, set to **Yes**. If this entry is not listed or a value is not specified, it defaults to **No**.
- **ShowParameters**. Yes or No. Custom components will use the value **Yes**, allowing the parameters to be modified by the user in Output Transformation Designer. The **ClassParm** value must be valid if ShowParameters is **Yes**.

12.1.2.2.4.1 Optional and Retired Elements

You may see the following elements in an existing components.xml file:

- **Type**. The default value component does not have to be specified. This is the only option available for custom components created by API users.
- **Icon**. No longer used.
- **Package**. No longer used.
- **Common Package**. No longer used.
- **Wizard**. For internal use only.
- **Flags**. No longer used.

12.1.3 Preparing a Custom Component for use in Projects

Before you can use the new custom component in a project, you must modify the Output Transformation Engine system configuration file (`.d2esys`) with information about your `components.xml` file. Within the system configuration file, add the full file path location of your `components.xml` file to the **ComponentDefinition** > **FileName** parameter and save your changes.

To make the custom component accessible to all projects in your environment, you must add the custom JAR file to the system classpath in addition to making the above changes to the system configuration.

Chapter 13

Working with JavaScript and the Script Component

The Script component provides users with a means of including scripts for interacting with key content in your projects such as document structure, page content, and extracted field data. API users can employ the component to add custom logic to their processes using JavaScript instead of writing, compiling, and managing custom Java-based components. At runtime, the component's script is executed once per page, following the same behavior as that of custom Java components.

JavaScript and EcmaScript

The terms JavaScript and EcmaScript are often used interchangeably. By definition, EcmaScript refers to the scripting language based on the international ECMA-262 specification.

JavaScript is an implementation of EcmaScript so online documentation, tutorials, and code samples exist using both names and either can be referenced. For more information on compatible versions of these scripting languages and some additional resources, see "JavaScript Compatibility" on page 72.

13.1 Configuring the Script Component

The Script component is available under the **Component** tab in the **Palette** and can be inserted into your project flows. It is a consumer type component so it must be positioned downstream from a parser. The Script component is also a producer type component and supports unlimited downstream consumers, so it is suitable for branding and routing logic. Multiple instances of the Script component can be used within a project.

There are two types of scripting: parameter scripting and component scripting.

Parameter Scripting

Parameter scripting allows the use of JavaScript to dynamically resolve the values of any configuration parameters. This is sometimes required when configuration options are dependent on some external criteria and cannot be set statically.

In addition to the standard wizards, menus, and embedded editors, most configuration parameters can also be accessed through the standard **Properties** window. Here, you can open a Property dialog box by clicking a parameter's respective ellipsis button. Then in the Property dialog box, you can activate the embedded **JavaScript Editor** from the drop-down menu.

Using the JavaScript Editor, you can edit your source code while the editor offers syntax highlighting and some basic code completion.

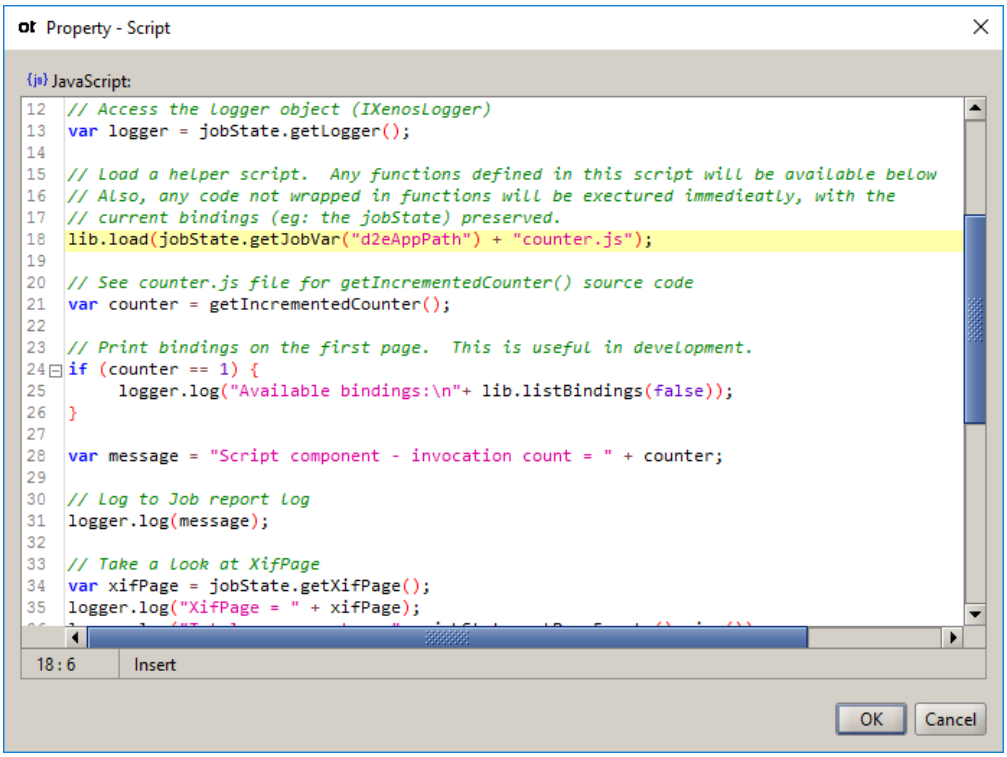


Figure 13-1: JavaScript Editor instance with Script component sample

Component Scripting

As an alternative to writing, compiling, and managing custom Java-based components, API users can use the Script component to add custom logic to their processes using JavaScript. The scripts can be used to interact with key content in your projects such as document structure, page content, and extracted field data.

At runtime, the component's script is executed once per page, following the same behavior as that of custom Java components. Many API users who create custom Java components in Output Transformation Engine will subclass one of these Java classes:

- `com.xenos.d2e.xdc.common.BufferedBaseComponent`
- `com.xenos.d2e.xdc.common.OneToOneComponent`

The Script component is an implementation of `com.xenos.d2e.xdc.common.BufferedBaseComponent` and JavaScript users can access all of the same functionality available in this Java class.

The component itself provides some additional convenience methods which are designed to be called from scripts, reducing the amount of scripting code that a user must write.

13.2 Event Processing and Routing

The Script component supports two main use cases:

- Content processing
- Document and page routing

In most situations, scripts are created for content processing, where they are required to examine and/or modify page content and perform other business logic within the existing document structure. These scripts do not usually need to modify the order of pages or filter out pages that should not be included in the generated output document.

In other cases, custom logic may be required to suppress pages or perform routing to different generators. These types of use cases require the script to interact with the flow of page events.

To support both of these types of use cases, the Script component has a configuration parameter called EventForwarding, which controls how XdcEvents are sent downstream by the Script component. By default, the Script component executes the user's script and then automatically passes the XdcEvents to all connected IXdcConsumer components downstream. This would be equivalent to adding `jobState.getComponent().sendAllEvents();` at the end of the script. In many cases, this is convenient as the scripting logic is used to extract data or manipulate document, while still retaining the original page order of the document. However, in some circumstances, a script may be used to break up a large document into smaller documents, suppress pages, or perform custom routing of pages to different generators. For these situations, EventForwarding should be set to Manual and page events should be sent downstream in the script by calling `jobState.getComponent().sendEvent()` or other such methods.

```
//"import" Java classes which will be constructed in this script
var XifDocument = com.xenos.d2e.transform.common.xif.XifDocument;
var XifEndOfDocument = com.xenos.d2e.transform.common.xif.XifEndOfDocument;

var logger = jobState.getLogger();

try {
        var scriptComponent  = jobState.getComponent();
        var afpGeneratorName = "AFP Generator";
        var pdfGeneratorName = "Accessible PDF Generator";
        var xifPage          = jobState.getXifPage();

        // Use a properties file which has properties in the format <CUSTOMER_ACCT_NUMBER> = AFP | PDF
        var customerProps = lib.loadProperties("${d2eAppPath}customers.properties", "customerProps");

        // Look at the current account number and look up the desired output type
        var currentAccountNumber = jobState.getXftField("Account Number");
        var customerOutputType   = customerProps.getProperty(currentAccountNumber);

        if (customerOutputType == "AFP") {
                // Send this page to the AFP generator to add to single file called "output.afp"
                scriptComponent.addToDocument("output.afp", afpGeneratorName, xifPage);
        }
        else if (customerOutputType == "PDF") {
                // Send this page to the PDF generator to add to a file called <ACCOUNT_NUMBER>.pdf
                // Document breaks are managed automatically by the scriptComponent.
                scriptComponent.addToDocument(currentAccountNumber+".pdf", pdfGeneratorName, xifPage);
        }
        else {
                // Something unexpected happened.
                logger.log("Could not resolve output type for customer [" + currentAccountNumber + "]");
        }

        logger.log("Page " + xifPage.getFilePageNumber() + ", Acct# " +  currentAccountNumber +
                ", Output type = " + customerOutputType);

}
catch (exception) {
        lib.logError("Error in script!", exception);
}
```

Figure 13-2: Sample code of an event routing instance

13.3 Writing Scripts

Scripts can be written at the parameter or component level, but share some common principles, functionality, and environment variables.

13.3.1 Standard Bindings

When JavaScript functions are executed, a set of variables called bindings are made available to the script as variables. These binding variables provide a mechanism for passing data and objects to and from the script. The bindings are Java objects that can be seamlessly accessed in the JavaScript code the same way you would interact with any JavaScript variable or object.

The standard set of bindings are described below. You can also learn more about these Java binding objects by referring to the corresponding Javadoc documentation in the help system.

13.3.1.1 jobState

The jobState object is the primary mechanism for interacting with the running process and underlying environment. It provides access to job variables and other stateful objects.

The underlying jobState Java object available to you in your script is dependent on the context on which the script is executed. This allows scripts to access state objects which are relevant for their application. For example, jobState will provide access to the current page object (XifPage) and any text fields that have been located.

There is a base Java class that defines methods available to all types of job states, as well as various subclasses that provide additional functionality. These subclasses inherit all methods and properties of the base class. Some code snippets showing sample usage are also shown below.

Content	Corresponding Java Class Name	Details
Default	`com.xenos.framework.util.jobstate.JobState`	The base job state. Provides access to properties such as job variables, job ID, active logger, and I/O utilities.
Output Transformation Designer	`com.xenos.framework.component.RunnableJobState`	In addition to the properties provided by the default job state, this Java class offers access to the JobTicket object for the running process.
Output Transformation Engine (Component level)	`com.xenos.d2e.jobstate.D2eJobState`	In addition to the properties provided by the default job state, this Java class offers access to the current component and any state events associated with the active page (field found events).
Output Transformation Engine (System level)	`com.xenos.d2e.jobstate.D2eProcessState`	Output Transformation Engine operates as a service and some system configuration parameters may use scripting. This system state object provides the basic job state properties.

```
//Access and use the active logger.  Expect output to be logged to the corresponding log for your job
var logger = jobState.getLogger();
logger.log("Here is a message for the log!");

try {
        //The job ID for the associated job.  Could be a Process Designer job, or Doc. Transform job
        var jobId = jobState.getJobId();

        //Every instance of JobState has a unique, numeric ID.  Useful for debugging, etc.
        var jobStateId = jobState.getJobStateId();

        //Access job variables which are strings.  Optionally provide a default value
        var defaultFileName = "defaultTestIndex.txt";
        var indexFileName = jobState.getJobVar("indexFileName"); // if not defined, returns empty string ""
        var indexFileName = jobState.getJobVar("indexFileName", defaultFileName);

        //Access job variables which are objects
        var customerObject = jobState.getJobVarObject("Customer");
        //Depending on what type of object it is, you can call the available methods.
        var customerName = customerObject.getName();

        //Access the IoUtility instance
        var ioUtility = jobState.getIoUtility();
        var inputStream = ioUtility.createInputStream("{NOCRLF}", indexFileName, "");

        //Replace job variables.  If variables are not defined, the place holders remain in the string
        var stringWithJobVars = "Here is a string that contains a job variables ${myJobVar} and ${myJobVar2}";
        var stringWithoutJobVars = jobState.replaceJobVariables(stringWithJobVariables);
}
catch (exception) {
        // Catch any possible exception
        logger.log("Error in script: " + exception);
}
```

Figure 13-3: Sample usage of default JobState

```
// Get a logger instance
var logger = jobState.getLogger();

try {
        // Importing classes: This method works for Java 6, 7 and 8
        var XDoc = com.xenos.framework.component.XDoc;

        //get the current JobTicket instance
        var jt = jobState.getJobTicket();

        //working with XDocs
        var xdoc = new XDoc("Hello World");
        jt.putInputMap("input", xdoc);

        var xdoc2 = jt.getInputXDocFor("input");
        var message = lib.getBytesAsString(xdoc2.toByteArray());
        logger.log("Value of input XDoc is " + message);

        //working with job variables
        jobState.setJobVar("foo", "bar");
        var foo = jobState.getJobVar("foo");
        logger.log("Value of foo is " + foo);
}
catch (exception) {
        // Catch any possible exception
        logger.log("Error in script: " + exception);
}
```

Figure 13-4: Sample usage of RunnableJobState

```
// Get a logger instance
var logger = jobState.getLogger();

try {
        // Get access to the JobRecord
        var jobRecord = jobState.getJobRecord();
        var userContext = jobRecord.getUserContext();

        // Access the active XDC project component (ex: AFP Parser, XFT, etc)
        var xdcComponent = jobState.getComponent();

        //Take a look at XifPage
        var xifPage = jobState.getXifPage();
        logger.log("XifPage = " + xifPage);
        logger.log("Total page events = " + jobState.getPageEvents().size());

        // Access the Xif document binders.
        var xifDocument = jobState.getXifDocument();  // the current document
        // End of current doc (if ended on current page), or end of previous document
        var xifEndDocument = jobState.getXifEndDocument();

        // For advanced XFT processing, access the XFT structures directly.
        var xftDocument = jobState.getXftDocument(); // Get the current XftDocument
        var xftSection  = jobState.getXftSection(); // Current (most recently processed) XFT section

        // For basic XFT processing, the job state provides access to field results
        // Look for a known XFT Text field called "Prepared For" in Section named "Heading"
        // This returns a String.  You can also omit the section parameter: jobState.getXftField("Prepared For");
        // If you want the XftTextFieldResult object for deeper analysis,
        // use the jobState.getXftFieldResult() methods.
        var textField = jobState.getXftField("Heading", "Prepared For");
        logger.log("Found text field: " + textField + "");

        // This field result returns this format, so let's chop off the "Prepared for: " prefix
        // "Prepared for: April Black 110-4297 Main St. Birmingham AL 35203"
        if (textField  &&  textField.contains("Prepared for:")) {
                var shortTextField = textField.substring(14);
                logger.log("  short text field: " + shortTextField);
        }
}
catch (exception) {
        // Catch any possible exception
        logger.log("Error in script: " + exception);
}
```

Figure 13-5: Sample usage of D2eJobState

13.3.1.2 lib

The lib binding provides convenient library methods for scripts to utilize. Unlike job state objects, there is only one library object available, which is an instance of the `com.xenos.framework.script.JavaScriptLib` object.

A significant feature of JavaScriptLib is the ability to import existing library scripts into new scripts. This is intended to allow users to build up a library of functions that they wish to reuse throughout a series of scripts.

```
// Load an external JavaScript file.  If this file contains functions, then code below this
// will be able to call those functions.  If the file contains JavaScript outside of
// functions, then that code will be executed.  All variables defined in that code would
// be available below.
lib.load("${projectDir}counter.js");

// Call the getIncrementedCounter() method from counter.js
var counter = getIncrementedCounter();

// For the first iteration, call lib.listBindings() to tell us all about the
// binding objects available to the script.  This is akin to a printHelp() function
if (counter == 1) {
        listOfBindingObjects = lib.listBindings(false);
        listOfBindingObjectsWithMethods = lib.listBindings(true);

        jobState.getLogger().log("Available bindings:\n"+ listOfBindingObjects);
}

// Load a properties file into an object
var props = lib.loadProperties("${projectDir}configFiles/config.properties");
var username = props.getProperty("username");

// Add all of these properties to my job variables
jobState.getJobVars().putAll(props);

// Access a input stream
var inputStream = lib.getFileInputStream("${projectDir}rawData.bin");

// Load a text file into a String using the default platform encoding
var textFile = lib.getFileAsString("${projectDir}customers.txt");
var fileByteArray = lib.getFileAsBytes("${projectDir}logo.jpg");

// Use lib to create Java arrays that are Nashorn (Java 8) and Rhino (Java 6 & 7) compatible
var byteArray = lib.newByteArray(100);
var objectArray = lib.newObjectArray(100);
var booleanArray = lib.newBooleanArray(100);

try {
        // code that throws an exception
}
catch (exception) {
        // Use lib to get info about the exception, including message and line number
        var details = lib.getExceptionDetails(exception);

        // Log various different types of messages, which will alter return codes
        lib.logWarning("This is a warning, increases return code to 4");
        lib.logWarning("This is a warning, increases return code to 4, including exception details", exception);
        lib.logError("This is an error, increases return code to 8");
        lib.logError("This is an error, increases return code to 8, including exception details", exception);
}
```

Figure 13-6: Sample usage of JavaScriptLib

```
function getIncrementedCounter() {

        // Get the existing variable called "counter".  The second argument of "0" (zero) is the
        // default value that will be set if the job variable does not yet exist.  So the first
        // time this function is called during a job, counter will be initialized to 0.
        // Subsequent calls will retrieve the incremented value.
        var counter = jobState.getJobVarObject("counter", 0);
        counter++;

        // We have to overwrite the existing JobVar because "counter" is a stored as a
        // java.lang.Integer, which is immutable.   This technique would be required for all
        // other primitive types like byte, char, short, long and also String.  When modifying
        // properties of a more complex Object the setJobVar() may not be required.
        jobState.setJobVar("counter", counter);
        return counter;
}
```

Figure 13-7: Contents of counter.js

13.3.1.3 result

The result binding is utilized in parameter scripts as a mechanism to return the desired value of the parameter. The value of the result is treated as a string and then converted to the native type defined for the parameter (i.e. boolean, int, float, etc.). Any library scripts that get called during lib.load() will also have the ability to update the result binding.

Note: The result binding is ignored in component scripting.

```
// Set a default result value first
result = false; // could use "false" too

//Get a logger instance
var logger = jobState.getLogger();

try {
        if (jobState.getJobVar("EnableBooleanParameter") == "true") {
                // Update the result
                result = true;
        }
}
catch (exception) {
        logger.log("Error in script: " + exception);
}

// Let the script end, the script engine will take the final value of the result binding
```

Figure 13-8: Sample usage of result binding

13.4 Exception Handling and Default Job States

Exception handling is demonstrated in the sample code shown throughout the JavaScript component chapter by means of `try` and `catch` statements. While the usage of these is optional, they are highly recommended because they give you full control over your own error handling logic.

When a `try/catch` statement is not present, any exceptions encountered in the script are caught by the script executor. The effect of this depends on the context; in a script component, the job is aborted with an error while in parameter scripts, the error handling is less predictable because parameters are evaluated in hundreds of situations through the OpenText Output Transformation Suite engine where error handling logic varies. Parameter scripts also have to consider that it is technically possible for parameter values to be requested when there is no active job or some other expected state objects not being populated. For instance, some parameters may be called during or prior to job initialization. In these situations, the jobState binding is available, but it may be a default job state object. The logger in the default job stats will log to the console `(standard out)`. When using `try/catch` statements, you can anticipate the use of default job states and react accordingly.

13.5 JavaScript Compatibility

JavaScript support in OpenText Output Transformation Suite utilizes the JVM's built-in JavaScript script engine. This means the underlying JavaScript engine depends on the actual JVM being used at runtime.

Native JavaScript support was first introduced in Java 6 with JSR-223. This implementation used a modified version of Mozilla Rhino 1.6R2, which supports the JavaScript 1.5/ECMA-262 Edition 3 specification.

In Java 8, the Rhino engine has been replaced with Nashorn (the German word for Rhino). Nashorn provides much better performance and supports newer versions of JavaScript and ECMA scripts. It is important to be aware that there are some compatibility differences between Rhino and Nashorn so some scripts that run on Rhino may not necessarily run on Nashorn and vice versa.

Fortunately, script writers can be aware of the incompatibilities and consciously write scripts that are compatible with both engine types. This is especially important for users building applications that will run on Java 6 or 7 today, but may be upgraded to Java 8+ in the future. For more information, see "Creating Rhino and Nashorn Compatible Scripts" on page 72.

Java Version	Script Engine	JavaScript/ECMA Script Version
Default	`com.xenos.framework.util.jobstate.JobState`	The base job state. Provides access to properties such as job variables, job ID, active logger, and I/O utilities.
Output Transformation Designer	`com.xenos.framework.component.RunnableJobState`	In addition to the properties provided by the default job state, this Java class offers access to the JobTicket object for the running process.

13.5.1 Creating Rhino and Nashorn Compatible Scripts

To help maintain compatibility between Rhino and Nashorn scripts, follow these basic tips:

- Avoid using Rhino's JavaAdapter
- Avoid the use of importPackage and importClass
- Use Java native types instead of JavaScript types
- Use the provided JavaScriptLib (lib binding) to create arrays, ex: `lib.newByteArray(size)`
- Inspect JavaScript exceptions caught in try/catch blocks using `lib.getExceptionDetails(jsException)`

For further information on migrating from Rhino to Nashorn and highlights of the technical incompatibilities, see:

https://wiki.openjdk.java.net/display/Nashorn/Rhino+Migration+Guide (https://wiki.openjdk.java.net/display/Nashorn/Rhino%2bMigration%2bGuide)

Supplementary information about these JavaScript specifications can be found through the following links:

Java Specification	Link
Oracle Nashorn	http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html
Mozilla JavaScript reference	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference
ECMA script specification	http://es5.github.io/

13.6 Printing Documents Within a Print File

A print file may contain many separate documents and the Script Component can group the pages within a document so they can be processed together. After the pages are processed, they can be sent to subsequent components in the workflow for further transformations.

To identify an individual document, you must create a document field definition with the XFT Field Technology component with a selector identifying the first page of each document. (For more information about the XFT Field Technology component, see *OpenText Embedded Output Transformation Engine - User Guide (VDTOTS-H-UTE)*.)

Then to group the pages together, a JavaScript array is created when a document is located. See the sample code of the xifDocument being put into an array and then placed into the jobState:

```
if (jobState.isDocBreak()
{ // found first page in document pageArray = []; // Array to hold xifDocument and
xifPages pageArray.push(jobState.getXifDocument()); jobState.setJobVar("pageArray",
pageArray); }
```

For the detection of pages, the array is retrieved from the jobState, and then the pages are added to the array before it is put back into the jobState. This sample code illustrates the retrieval of the array and the addition of the pages prior to the array being returned into the jobState:

```
if (xifPage)
{var pageArray = jobState.getJobVarObject("pageArray"); pageArray.push(xifPage);
jobState.setJobVar("pageArray", pageArray); }
```

When the end of the document is reached, the pages can be processed. This sample code illustrates processing all pages held in the array:

```
if (jobState.getXifEndDocument() || jobState.isEOF()) { // found end of documentvar
pageArray = jobState.getJobVarObject("pageArray");// process each page in the arrayvar
xifDocument = pageArra[0];for (i = 1; i < pageArray.length; i++)
```

```
{var xifPage = pageArray[i]; ... }
}
```

After all pages are processed, they can be passed to the next component in the workflow. You can perform this by passing the following code sample:

```
var scriptComponent = jobState.getComponent();scriptComponent.sendEvent(xifPage);
```

After the last page in the final document is sent to the next component, you must send the end-of-document event.

```
scriptComponent.sendEvent(jobState.getXifEndDocument());
```

As an example of the type of further processing that can be performed in your project, you can use an additional script to add barcodes to each page in a document. The following code sample shows how this can be done:

```
try {
 // Access System objects.
 var scriptComponent  = jobState.getComponent();
 var xifPage = jobState.getXifPage();  // the current page
 var xifDocument = jobState.getXifDocument(); // the current document

 // in XFT the document must have a selector to indicate the first page
 // of the document. User must enable start identification.

 if (jobState.getXifEndDocument() || jobState.isEOF()) {
  // end of document
  lib.logInfo("script - perform end of document processing");
  var pageArray = jobState.getJobVarObject("pageArray");
  // process each page in the array
  for (i = 0; i < pageArray.length; i++) {
   var xPage = pageArray[i];
   if (i == 0) {
    var xifDocument = xPage;
    lib.logInfo("script - found XifDocument " + xifDocument);
   else {
    lib.logInfo("script - found XifPage " + xPage);
    var digit1 = "0";  // inserts
    var digit2 = xPage.documentPageNumber;
    var digit3 = "0";

    if (i == (pageArray.length - 1)) {
     // last page
     digit3 = "1";

    var barcodeStr = digit1 + digit2 + digit3;

    // build an args object (see http://www.w3schools.com/js/js_objects.asp)
    var barcodeArgs = {
     xifPage  : xPage,
     x        : (7.5 * 72000),  // top left in 72000 DPI
     y        : (0.2 * 72000),  // top left in 72000 DPI
     width    : (1 * 72000),    // width in 72000 DPI
     height   : (0.3 * 72000),  // height in 7200 DPI
     data     : barcodeStr,     // data to encode
     mfctColor: com.xenos.d2e.transform.common.mfct.MfctColor.BLACK, // MfctColor to use
     rotation : 90, // Rotation (0, 90, 180, 270)
     type     : com.xenos.d2e.transform.common.xif.barcode.BarCodeSymbol.TYPE_CODE39 //
3 of 9
     // NOTE: other types of linear barcodes are availble, refer the Javadoc for the
BarCodeSymbol class.
    ;
    addLinearBarcode(barcodeArgs);
   scriptComponent.sendEvent(xPage);
   lib.logInfo("script - sent page event - " + xPage);
   // end of for loop
```

```
  scriptComponent.sendEvent(jobState.getXifEndDocument());
  lib.logInfo("script - sent XifEndDocument event")

 if (jobState.isDocBreak()) {
        // found the document
  lib.logInfo("script - isDocBreak - Perform Start of document processing");
  pageArray = [];  // array to hold xifDocument/xifPages
  lib.logInfo("script - found XifDocument");
  // add XifDocument to array
  pageArray.push(jobState.getXifDocument());
  jobState.setJobVar("pageArray", pageArray);

 if (xifPage) {
  lib.logInfo("script - found XifPage");
  var pageArray = jobState.getJobVarObject("pageArray");
  // add XifPage to array
  pageArray.push(xifPage);
  jobState.setJobVar("pageArray", pageArray);

 if (jobState.isEOF()) {
  //Process all events
  lib.logInfo("script - isEOF - Perform End of File processing" );
  scriptComponent.sendAllEvents();
 }

catch (exception) {
 lib.logError("Error in script!", exception);
 }

// Function to add linear barcodes
function addLinearBarcode(args) {
 var linearArgs = new com.xenos.d2e.transform.common.xif.barcode.XifLinear(args.type);

 var fes = jobState.getJob().getFontEnvironmentSession();
 if (fes) {
  var hriFont = fes.resolveAwtFont("ocra10");
  if (hriFont) {

linearArgs.setHRI(com.xenos.d2e.transform.common.xif.barcode.XifBarcodeAttributeBase.HRIB
ELOW);
           linearArgs.setHriFont(hriFont);
            linearArgs.setUseBarcodeSpecificHRIFont(true);

 var xifBarcode = new com.xenos.d2e.transform.common.xif.XifBarcodeElement(

  linearArgs,
  args.x,        // top left in 72000 DPI
  args.y,        // top left in 72000 DPI
  args.width,    // width in 72000 DPI
  args.height,   // height in 7200 DPI
  args.data,     // Data to encode
  args.mfctColor,// MfctColor to use
  args.rotation);// Rotation (0, 90, 180, 270)

   args.xifPage.addBarcode(xifBarcode);
}
```