Informatica (Version 9.5.1)

Data Transformation User Guide
This product includes software licensed under the Academic Free License (http://www.opensource.org/licenses/afl-3.0.php), the Common Development and Distribution License (http://www.opensource.org/licenses/cddl1.php), the Common Public License (http://www.opensource.org/licenses/cpl1.0.php), the Sun Binary Code License Agreement Supplemental License Terms, the BSD License (http://www.opensource.org/licenses/bsd-license.php) the MIT License (http://www.opensource.org/licenses/mit-license.php) and the Artistic License (http://www.opensource.org/licenses/artistic-license-1.0).

This product includes software copyright © 2003-2006 Joe WaInes, 2006-2007 XStream Committers. All rights reserved. Permissions and limitations regarding this software are subject to terms available at http://xstream.codehaus.org/license.html. This product includes software developed by the Indiana University Extreme! Lab. For further information please visit http://www.extreme.indiana.edu/.


This Software is protected by U.S. Patent Numbers 5,794,246; 6,014,670; 6,016,501; 6,029,178; 6,032,158; 6,035,307; 6,044,374; 6,092,086; 6,208,990; 6,339,775; 6,640,226; 6,789,096; 6,820,077; 6,823,373; 6,850,947; 6,895,471; 7,117,215; 7,162,643; 7,243,110; 7,254,590; 7,281,001; 7,421,458; 7,496,588; 7,523,121; 7,584,422; 7676516; 7,720,842; 7,721,270; and 7,774,791, international Patents and other Patents Pending.

DISCLAIMER: Informatica Corporation provides this documentation "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of noninfringement, merchantability, or use for a particular purpose. Informatica Corporation does not warrant that this software or documentation is error free. The information provided in this software or documentation may include technical inaccuracies or typographical errors. The information in this software and documentation is subject to change at any time without notice.

NOTICES

This Informatica product (the "Software") includes certain drivers (the "DataDirect Drivers") from DataDirect Technologies, an operating company of Progress Software Corporation ("DataDirect") which are subject to the following terms and conditions:

1. THE DATADIRECT DRIVERS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.
2. IN NO EVENT WILL DATADIRECT OR ITS THIRD PARTY SUPPLIERS BE LIABLE TO THE END-USER CUSTOMER FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL OR OTHER DAMAGES ARISING OUT OF THE USE OF THE ODBC DRIVERS, WHETHER OR NOT INFORMED OF THE POSSIBILITIES OF DAMAGES IN ADVANCE. THESE LIMITATIONS APPLY TO ALL CAUSES OF ACTION, INCLUDING, WITHOUT LIMITATION, BREACH OF CONTRACT, BREACH OF WARRANTY, NEGLIGENCE, STRICT LIABILITY, MISREPRESENTATION AND OTHER TORTS.

Part Number: DT-DPT-95100-0001
# Table of Contents

**Preface** ................................................................. xiii

**Chapter 1: Introduction to Data Transformation** ......................................................... 1

- Data Transformation Overview ................................................................. 1
- Data Transformation Process Architecture .................................................. 2
- Data Transformation Components .............................................................. 3
- Data Transformation Options ..................................................................... 3

**Chapter 2: Data Processor Transformation** ......................................................... 4

- Data Processor Transformation Overview ................................................... 4
- Data Processor Transformation Views .......................................................... 5
- Data Processor Transformation Ports .......................................................... 5
  - Data Processor Transformation Input Ports ............................................... 6
  - Data Processor Transformation Output Ports ............................................ 7
  - Relational Output ..................................................................................... 8
  - Pass-Through Ports .................................................................................. 10
- Startup Component .................................................................................... 10
- Schema References ..................................................................................... 11
- Data Processor Transformation Settings ..................................................... 11
  - Character Encoding .................................................................................. 11
  - Rules and Guidelines for Character Encoding .......................................... 14
  - Output Settings ....................................................................................... 14
  - Processing Settings ............................................................................... 15
  - XML Output Configuration ..................................................................... 15
- Events .......................................................................................................... 17
  - Event Types .............................................................................................. 17
  - Data Processor Events View ................................................................... 18
- Logs ............................................................................................................... 18
  - Design-Time Event Log ........................................................................... 19
  - Run-Time Event Log ............................................................................... 19
  - Viewing an Event Log in the Data Processor Events View ....................... 20
  - User Log ................................................................................................. 20
- Data Processor Transformation Development .............................................. 20
  - Create the Data Processor Transformation .............................................. 20
  - Select the Schema Objects ...................................................................... 21
  - Step 3. Create Objects .............................................................................. 21
  - Step 4. Create the Ports ............................................................................ 22
  - Step 5. Testing the Transformation .......................................................... 23
Chapter 9: Formats. .......................................................... 79
Formats Overview. ........................................................ 79
Standard Format Properties. ............................................ 80
Format Component Reference. ........................................... 80
  BinaryFormat. ........................................................ 80
  CustomFormat. ....................................................... 81
  HtmlFormat. ........................................................ 82
  RtfFormat. ........................................................ 83
  TextFormat. ........................................................ 84
  XmlFormat. ........................................................ 85
Delimiters Component Reference. ....................................... 86
  CommaDelimited. .................................................... 87
  Delimiter. .......................................................... 87
  DelimiterHierarchy. ................................................ 88
  EnclosingDelimiters. ............................................... 88
  HL7. .............................................................. 89
  Positional. ......................................................... 89
  PostScript. ......................................................... 90
  RTF. ............................................................. 90
  SGML. ............................................................ 90
  SpaceDelimited. ................................................... 90
  TabDelimited. ..................................................... 90
Format Preprocessor Component Reference. .......................... 91
  HtmlProcessor. ................................................... 91
  RtfProcessor. ..................................................... 91
## Chapter 10: Data Holders

- Data Holders Overview .................................................. 92
- XML Schemas .......................................................... 92
  - About Schemas ....................................................... 93
  - How to Create Schemas ................................................. 94
  - Schema Encoding ..................................................... 94
  - Included Schema Files .................................................. 94
  - Namespaces ........................................................... 94
  - Mixed Content ........................................................ 94
  - Unsupported Schema Features ............................................. 95
  - Precision of Numerical Data ............................................... 96
- Using a Schema to Map Anchors ............................................... 96
  - IntelliScript Representation of Data Holders ....................... 96
  - Mapping Mixed Content .................................................. 96
  - Mapping XSI Types ..................................................... 97
- Generating Valid XML ...................................................... 97
  - Role of Schemas in Parsing ............................................... 98
  - Role of Schemas in Serialization and Mapping ................................ 98
  - Variables .............................................................. 99
    - Creating a User-Defined Variable ........................................... 99
    - System Variables ...................................................... 99
    - Mapping Anchors to Variables ........................................... 102
    - Using Variables in Actions ............................................... 102
    - Initializing Variables at Runtime ........................................ 102
  - Variable Component Reference ............................................... 102
    - Variable ........................................................... 102
- Multiple-Occurrence Data Holders ............................................. 103
  - Attributes .......................................................... 104
  - Indexing ........................................................... 104
  - Destroying the Occurrences .............................................. 104

## Chapter 11: Anchors

- Anchors Overview ........................................................ 106
  - Marker and Content Anchors .............................................. 106
  - Other Anchor Types ................................................... 106
  - How Anchors and Delimiters Work Together ............................ 107
- Mapping Content Anchors to Data Holders ............................. 107
  - Mapping to Variables ................................................... 108
  - Mapping to Multiple-Occurrence Data Holders ........................ 108
  - Mapping to Mixed-Content Elements ........................................ 108
- Defining Anchors ........................................................ 108
  - Where to Define Anchors ................................................ 108
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence of Anchors.</td>
<td>108</td>
</tr>
<tr>
<td>Adding a Marker or Content Anchor.</td>
<td>109</td>
</tr>
<tr>
<td>Defining an Anchor.</td>
<td>109</td>
</tr>
<tr>
<td>Standard Anchor Properties.</td>
<td>110</td>
</tr>
<tr>
<td>How a Parser Searches for Anchors.</td>
<td>111</td>
</tr>
<tr>
<td>Search Phases.</td>
<td>111</td>
</tr>
<tr>
<td>Search Scope and Search Criteria.</td>
<td>112</td>
</tr>
<tr>
<td>Adjusting the Search Phase.</td>
<td>113</td>
</tr>
<tr>
<td>Adjusting the Search Scope.</td>
<td>113</td>
</tr>
<tr>
<td>Adjusting the Search Criteria.</td>
<td>115</td>
</tr>
<tr>
<td>Using Data Types to Narrow the Search Criteria.</td>
<td>116</td>
</tr>
<tr>
<td>Anchors that Contain Nested Anchors.</td>
<td>117</td>
</tr>
<tr>
<td>Anchor Component Reference.</td>
<td>117</td>
</tr>
<tr>
<td>Alternatives.</td>
<td>118</td>
</tr>
<tr>
<td>Content.</td>
<td>120</td>
</tr>
<tr>
<td>DelimitedSections.</td>
<td>123</td>
</tr>
<tr>
<td>EmbeddedParser.</td>
<td>125</td>
</tr>
<tr>
<td>EnclosedGroup.</td>
<td>127</td>
</tr>
<tr>
<td>ExtractRecord.</td>
<td>129</td>
</tr>
<tr>
<td>FindReplaceAnchor.</td>
<td>129</td>
</tr>
<tr>
<td>Group.</td>
<td>132</td>
</tr>
<tr>
<td>Marker.</td>
<td>134</td>
</tr>
<tr>
<td>RepeatingGroup.</td>
<td>137</td>
</tr>
<tr>
<td>StructureDefinition.</td>
<td>140</td>
</tr>
<tr>
<td>Searcher Component Reference.</td>
<td>144</td>
</tr>
<tr>
<td>AttributeSearch.</td>
<td>144</td>
</tr>
<tr>
<td>LearnByExample.</td>
<td>145</td>
</tr>
<tr>
<td>NewlineSearch.</td>
<td>146</td>
</tr>
<tr>
<td>OffsetSearch.</td>
<td>146</td>
</tr>
<tr>
<td>PatternSearch.</td>
<td>146</td>
</tr>
<tr>
<td>SegmentSearch.</td>
<td>147</td>
</tr>
<tr>
<td>TextSearch.</td>
<td>147</td>
</tr>
<tr>
<td>TypeSearch.</td>
<td>148</td>
</tr>
<tr>
<td>Anchor Subcomponent Reference.</td>
<td>148</td>
</tr>
<tr>
<td>AllStructure.</td>
<td>148</td>
</tr>
<tr>
<td>AllStructureLocal.</td>
<td>149</td>
</tr>
<tr>
<td>ChoiceStructure.</td>
<td>150</td>
</tr>
<tr>
<td>ChoiceStructureLocal.</td>
<td>150</td>
</tr>
<tr>
<td>Connect.</td>
<td>151</td>
</tr>
<tr>
<td>EmbeddedStructure.</td>
<td>152</td>
</tr>
<tr>
<td>RecordStructure.</td>
<td>152</td>
</tr>
<tr>
<td>RecordStructureLocal.</td>
<td>153</td>
</tr>
<tr>
<td>SequenceStructure.</td>
<td>154</td>
</tr>
</tbody>
</table>
Chapter 12: Transformers .......................................................... 156

Transformers Overview ....................................................... 156
Defining Transformers ..................................................... 156
    Using Transformers in Anchors ............................................ 156
Sequences of Transformers .............................................. 157
Default Transformers ................................................... 157
    Using Transformers as Document Processors ................................... 157
    Using Transformers in Serialization Anchors .................................... 158
    Using Transformers in Actions ............................................. 158
Standard Transformer Properties .............................................. 158
Transformer Component Reference ............................................ 158
    AbsURL ........................................................... 159
    AddEmptyTagsTransformer .............................................. 159
    AddString .......................................................... 160
    Base64Decode ....................................................... 160
    Base64Encode ....................................................... 161
    BidiConvert ......................................................... 161
    CDATADecode ....................................................... 162
    CDATAEncode ....................................................... 162
    ChangeCase ........................................................ 163
    CreateGuid ......................................................... 164
    CreateUUID ......................................................... 164
    DateFormatICU ...................................................... 164
    Dos96HebToAscii ..................................................... 167
    DynamicTable ....................................................... 167
    EbcdicToAscii ....................................................... 167
    EDIFACTValidation .................................................... 167
    EncodeAsUrl ........................................................ 168
    Encoder ........................................................... 168
    ExternalTransformer ................................................... 169
    FormatNumber ....................................................... 170
    FromFloat .......................................................... 171
    FromInteger ......................................................... 171
    FromPackDecimal ..................................................... 172
    FromSignedDecimal ................................................... 172
    hebrewBidi ......................................................... 173
    HebrewDosToWindows .................................................. 173
    HebrewEBCDICOldCodeToWindows ......................................... 173
    hebUniToAscii ....................................................... 173
    hebUtf8ToAscii ....................................................... 173
    HtmlEntitiesToASCII ................................................... 174
Chapter 13: Actions ................................................................. 192

Actions Overview ............................................................. 192
How Actions Work ............................................................. 192
Comparison Between Actions and Transformers ......................... 193
Defining Actions ............................................................... 193

Standard Action Properties .................................................. 193

Action Component Reference ................................................ 194
AddEventAction ............................................................... 194
AggregateValues ............................................................. 195
AppendListItems ............................................................... 197
AppendValues ................................................................. 198
CalculateValue ............................................................... 199
CombineValues ............................................................... 201
CreateList .................................................................... 202
CustomLog .................................................................... 203
Chapter 15: Mappers. ............................................................ 243
Creating a Mapper. ........................................................... 243
Components Nested within a Mapper. ........................................... 243
Mapper Example. ........................................................ 244
Source XML. ............................................................ 244
Output XML. ........................................................... 244
Mapper Configuration. .................................................. 245
Standard Mapper Properties. ................................................. 245
Mapper Component Reference. ............................................... 246
Mapper. ............................................................ 246
Mapper Anchor Component Reference. .......................................... 247
AlternativeMappings. ................................................... 248
EmbeddedMapper. .................................................... 249
GroupMapping. ....................................................... 250
RepeatingGroupMapping. ................................................ 251

Chapter 16: Locators, Keys, and Indexing. ........................................ 253
Overview of Locators, Keys, and Indexing. ........................................ 253
Example of Locators. ...................................................... 254
Input and Output. ...................................................... 254
Incorrect Solution. ..................................................... 254
Correct Solution. ...................................................... 255
Example of Indexing by Key. ................................................. 255
Input. ............................................................. 255
Output. ............................................................ 256
Outline of the Transformation Approach. ....................................... 256
Mapper Configuration. .................................................. 256
Use of Indexing. ...................................................... 258
Source and Target Properties. ................................................ 258
Source Property. ...................................................... 259
Target Property. ...................................................... 262
Standard Locator and Key Properties. .......................................... 264
Locator and Key Component Reference. .......................................... 264
Key. ............................................................. 264
Locator. ............................................................ 266
LocatorByKey. ....................................................... 267
LocatorByOccurrence. .................................................. 268

Chapter 17: Streamers. .......................................................... 269
Streamers Overview. ...................................................... 269
Chapter 19: Custom Script Components ....................................................... 311

Custom Script Components Overview ....................................................... 311
Custom Component Example ................................................................. 311
Custom Component Properties ............................................................... 312
Developing a Custom Component in Java ................................................ 312
  Java Interface Example ................................................................. 313
  Sample Custom Java Components ...................................................... 313
Developing a Custom Component in C or C++ ........................................ 314
  C/C++ Property Value Length Limitation ............................................ 315
  C/C++ Interface Example ............................................................... 315
  Sample Custom C/C++ Components .................................................. 315
Configuring a Custom Component .......................................................... 316
  Sample Scripts Containing Custom Components ................................... 317

Index ... .................................................................................................... 318
Preface

The *Informatica Data Processor Transformation Guide* is written for developers, analysts, and other users who are responsible for designing and implementing transformations. The book explains how to design, configure, test, and deploy transformations. It contains detailed reference sections documenting the transformation components and their properties.

This book assumes that you have a basic knowledge of how to use Informatica Developer. It also assumes that you understand XML, schemas, and basic programming techniques.
Chapter 1

Introduction to Data Transformation

This chapter includes the following topics:
- Data Transformation Overview, 1
- Data Transformation Process Architecture, 2
- Data Transformation Components, 3
- Data Transformation Options, 3

Data Transformation Overview

Data Transformation is an application that processes complex files, such as messaging formats, HTML pages and PDF documents. Data Transformation also transforms formats such as ACORD, HIPAA, HL7, EDI-X12, EDIFACT, AFP, and SWIFT.

Data Transformation installs by default when you install Informatica Developer (the Developer tool). You can define a Data Processor transformation to transform complex files in a mapping. When you run a mapping with the Data Processor transformation, the Data Integration Service calls the Data Transformation Engine to process the data.

The Data Transformation application has the following elements:

Data Processor transformation
A transformation that processes complex files in a mapping. Define Data Transformation script or XMap objects in the Developer tool to process the data. You can include the transformation in an SQL data service mapping, web service, or mapping profile.

Data Transformation service
A set of Data Transformation objects that you can export from Data Processor transformation and run standalone. You export a service to a Data Transformation repository and run the service from there.

Data Transformation repository
A directory that stores executable services that you export from a Data Processor transformation or that you deploy from the Data Transformation Studio. The repository directory name is ServiceDB.

Data Processor Engine
A processor that runs objects in the Data Processor transformation or services that you create in Data Transformation Studio.

Data Transformation Studio
A visual editor to design and configure Data Transformation services. You can design a Data Transformation service in the Data Transformation Studio instead of a Data Processor transformation. Deploy the project to the Data Transformation repository as a runnable service.
Data Transformation Process Architecture

You must install Data Transformation in order to configure and run a Data Processor transformation in the Developer tool. The Data Processor transformation can contain multiple scripts or XMap objects to transform complex files. The Data Transformation Engine runs the scripts or XMaps to transform the data. You can use a Data Processor transformation in a data service, web service, or profile.

The following figure shows the components in the Data Transformation application and the components that you use to create the same functionality in the Developer tool:

1. Create a Data Processor transformation in the Developer tool. Save the transformation in the Model repository.
2. Export the Data Processor transformation as a Data Transformation service. Export the service to the Data Transformation repository. You can run the service from the repository.
3. You can also create a Data Transformation project in the Data Transformation Studio that performs the same functions as the Data Processor transformation. Deploy the project as a service to the Data Transformation repository.
4. You can deploy an application that contains a Data Processor transformation to a Data Integration Service.
5. The Data Integration Service runs the application and calls the Data Processor Engine to process the transformation logic.
6. The Data Processor Engine also runs services from the Data Transformation repository.
Data Transformation Components

When you define a Data Transformation service or a Data Processor transformation, you can combine multiple components to transform the data.

Data Transformation has the following types of components that transform data:

**Parser**
Converts source documents to XML. The input can have any format. The output of a parser is XML.

**Serializer**
Converts an XML file to another document. The output can be any format.

**Mapper**
Converts an XML source document to another XML document.

**XMap**
Converts an XML source document to another XML document. XMap has the same functionality as Mapper, but you can use a grid in the Developer tool to define the mapping. You cannot define an XMap in the Data Transformation Studio.

**Transformer**
Modifies data in any format. Adds, removes, converts, or changes text. Use transformers with a parser, mapper, or serializer. You can also run a transformer as a standalone component.

**Streamer**
Splits large input documents, such as multiple gigabyte data streams, into segments. The streamer splits documents that have multiple messages or multiple records in them.

Data Transformation Options

You can choose several options to run Data Transformation. You can run Data Transformation as a stand-alone product or you can run Data Transformation services from a transformation in a mapping.

You must have one of the following options to use Data Transformation:

**Data Transformation Option**
Allows full use of Data Transformation, including the Data Transformation Studio and document preprocessors. Document preprocessors transform binary files such as PDF files or Word documents to text. With a Data Transformation Option, you can purchase libraries of components such as parsers, serializers, and XML schemas that you can use to process industry standard application messages.

**Unstructured Data Option**
Runs Data Transformation services in PowerCenter or in Data Services. The Unstructured Data Option includes data preprocessors, but no component libraries.

**Advanced XML Option**
Runs Data Transformation services in PowerCenter or in data services. The Advanced XML Option does not include data preprocessors or component libraries.
Chapter 2

Data Processor Transformation

This chapter includes the following topics:

- Data Processor Transformation Overview, 4
- Data Processor Transformation Views, 5
- Data Processor Transformation Ports, 5
- Startup Component, 10
- Schema References, 11
- Data Processor Transformation Settings, 11
- Events, 17
- Logs, 18
- Data Processor Transformation Development, 20
- Data Processor Transformation Import and Export, 23

Data Processor Transformation Overview

The Data Processor transformation processes unstructured and semi-structured file formats in a mapping. Configure the transformation to process messaging formats, HTML pages, XML, and PDF documents. Transform structured formats such as ACORD, HIPAA, HL7, EDI-X12, EDIFACT, AFP, and SWIFT.

For example, you might receive customer invoices in Microsoft Word files. You configure a Data Processor transformation to parse the data from each word file. Extract the customer data to a Customer table. Extract order information to an Orders table.

When you create a Data Processor transformation, you define XMap objects or script objects to transform the data. An XMap transforms an input XML document into an output XML document of another structure. A Script can parse source documents to XML, convert XML to other file formats, or map an XML document to another XML document of another format.

Define Scripts in the Data Processor transformation Script editor. You can define the following types of scripts:

- Parser. Converts source documents to XML. The output of a parser is always XML. The input can have any format, such as text, HTML, Word, PDF, or HL7.

- Serializer. Converts an XML file to an output document of any format. The output of a serializer can be any format, such as a text document, an HTML document, or a PDF.

- Mapper. Converts an XML source document to another XML structure or schema. You can transform the same XML documents as in an XMap.
Transformer. Modifies the data in any format. Adds, removes, converts, or changes text. Use Transformers with a parser, Mapper, or Serializer. You can also run a transformer as stand-alone component.

Streamer. Splits large input documents, such as multi-gigabyte data streams, into segments. The streamer processes documents that have multiple messages or records in them, such as HIPAA or EDI files.

Data Processor Transformation Views

The Data Processor transformation has multiple views that you access when you configure the transformation and run it in the Developer tool.

Some of the Data Processor transformation views do not appear in the Developer tool by default. To change the views for the transformation, click Window &gt; Show View &gt; Other &gt; Informatica. Select the views you want to see.

The Data Processor transformation has the following fixed views:

- **Overview view**
  Configure ports and define the startup component.

- **References view**
  Add or remove schemas from the transformation.

- **Settings view**
  Configure transformation settings for encoding, output control, and XML generation.

- **Objects view**
  Add, modify, or delete script and XMap objects from the transformation.

You can also access the following views for the Data Processor transformation:

- **Data Processor Hex Source view**
  Shows an input document in hexadecimal format.

- **Data Processor Events view**
  Shows information about events that occur when you run the transformation in the Developer tool. Shows initialization, execution, and summary events.

- **Script Help view**
  Shows context-sensitive help for the Script editor.

- **Data Viewer view**
  View example input data, run the transformation, and view output results.

Data Processor Transformation Ports

Define the Data Processor transformation ports on the transformation Overview view.

A Data Processor transformation can read input from a file, a buffer, or a streamed buffer. You can use a flat file reader as a buffer to read an entire file at one time. You can also read an input file from a database.

The output ports that you create depend on whether you want to return a string, complex files, or rows of relational data from the transformation.
Data Processor Transformation Input Ports

When you create a Data Processor transformation, the Developer tool creates a default input port. When you define an additional input port in a script startup component, the Developer tool creates an additional input port in the transformation.

The input type determines the type of data that the Data Integration Service passes to the Data Processor transformation. The input type determines whether the input is data or a source file path.

Configure one of the following input types:

**Buffer**

The Data Processor transformation receives rows of source data in the Input port. Use the buffer input type when you configure the transformation to receive data from a flat file or from an Informatica transformation.

**File**

The Data Processor transformation receives the source file path in the Input port. The Data Processor startup component opens the source file. Use the file input type to parse binary files such as Microsoft Excel or Microsoft Word files. You can also use the File input type for large files that might require a lot of system memory to process with a buffer input port.

**Service Parameter**

The Data Processor transformation receives values to apply to variables in the service parameter ports. When you choose the variables to receive input data, the Developer tool creates a service parameter port for each variable.

**Output_Filename**

When you configure the default output port to return a file name instead of row data, the Developer tool creates an Output_Filename port. You can pass a file name to the Output_Filename port from a mapping.

When you define an input port you can define the location of the example input file for the port. An example input file is a small sample of the input file. Reference an example input file when you create scripts. You also use the example input file when you test the transformation in the Data Viewer view. Define the example input file in the Input Location field.

**Service Parameter Ports**

You can create input ports that receive values for variables. The variables can contain any datatype such a string, a date, or a number. A variable can also contain a location for a source document. You can reference the variables in a Data Processor component.

When you create an input port for a variable, the Developer tool shows a list of variables that you can choose from.

**Creating Service Parameter Ports**

You can create input ports that receive values for variables. You can also remove the ports that you create from variables.

1. Open the Data Processor transformation Overview view.
2. Click **Choose**.
   The Developer tool displays a list of variables and indicates which variables already have ports.
3. Select one or more variables.
   The Developer tool creates a buffer input port for each variable that you select. You cannot modify the port.
4. To remove a port that you create from a variable, disable the selection from the variable list. When you disable the selection, the Developer tool removes the input port.
Data Processor Transformation Output Ports

The Data Processor transformation has one output port by default. If you define additional output ports in a Script, the Developer tool adds the ports to the Data Processor transformation. You can create groups of ports if you configure the transformation to return relational data. You can also create service parameter ports and pass-through ports.

Default Output Port

The Data Processor transformation has one output port by default. When you create relational output, you can define groups of related output ports instead of the default output port. When you define an additional output port in a script component, the Developer tool adds an additional output port to the transformation.

Configure one of the following output types for a default output port:

Buffer

The Data Processor transformation returns XML through the Output port. Choose the Buffer file type when you parse documents or when you map XML to other XML documents in the Data Processor transformation.

File

The Data Integration Service returns an output file name in the Output port for each source instance or row. The Data Processor transformation component writes the output file instead of returning data through the Data Processor transformation output ports.

When you select a File output port, the Developer tool creates an Output_Filename input port. You can pass a file name into the Output filename port. The Data Processor transformation creates the output file with a name that it receives in this port.

If the output file name is blank, the Data Integration Service returns a row error. When an error occurs, the Data Integration Service writes a null value to the Output port and returns a row error.

Choose the File output type when you transform XML to a binary data file such as a PDF file or a Microsoft Excel file.

Relational Output Ports

You can return relational output from the Data Processor transformation. If a component returns XML, you create groups of output ports by mapping nodes from the XML output to groups of ports. You can pass the data from the ports to another transformation in a mapping.

To return relational groups of ports, enable relational output from the Overview view. The Developer tool removes the default output port on the view.

Select Output Mapping. The Ports panel appears in the Overview view.

The following figure shows the Ports panel:
The XML output schema appears in the Hierarchical Output area of the panel. Create ports in the Relational Ports area of the panel and map nodes from the schema to the ports. You can also drag the pointer from a node in the schema to an empty field in the Ports area to create a port. When you map a node from the output schema to a port, the Developer tool shows a link between them.

Relational Output

When you configure relational output, you can configure a separate output group for each multiple-occurring input node. You can also create groups that contain denormalized data. You can pivot multiple-occurring elements and limit the number of occurrences in an output group.

Normalized Relational Output

When you create normalized output data, the data values do not repeat in an output group. You create a one-to-one relationship between the hierarchy levels in the XML hierarchy and the output groups of ports.

When the XML hierarchy contains a Departments parent hierarchy level and an Employees child hierarchy level, you might create the following groups of ports:

```
Departments
    Department_Key
    Department_ID
    Department_Name

Employees
    Department_Key
    Employee_ID
    Employee_Name
```

The Department_Key is a generated key that relates the Employees output group to a Department group.
**Pivoted Relational Output**

You can include a specific number of multiple-occurring elements in an output group.

To pivot multiple-occurring elements, map the multiple-occurring child element to the parent group of output ports. The Developer tool prompts you to define the number of child elements to include in the parent.

The following example shows two instances of Employee_ID in the Departments parent group:

```
Departments
  Department_ID
  Department_Name
  Employee_ID1
  Employee_ID2
```

**Denormalized Relational Output**

You can denormalize relational output. When you denormalize the output data, the element values from the parent group repeat for each child element.

To denormalize output data, map nodes from the parent hierarchy level to the child group of output ports.

The following example shows the Department_ID and the Department_Name in the Employees output group:

```
Employees
  Department_ID
  Department_Name
  Employee_ID
  Employee_Name
```

Department_ID and Department_Name repeat for each employee in the department:

<table>
<thead>
<tr>
<th>Department_ID</th>
<th>Department_Name</th>
<th>Employee_ID</th>
<th>Employee_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Accounting</td>
<td>56500</td>
<td>Kathy Jones</td>
</tr>
<tr>
<td>100</td>
<td>Accounting</td>
<td>56501</td>
<td>Tom Lyons</td>
</tr>
<tr>
<td>100</td>
<td>Accounting</td>
<td>56509</td>
<td>Bob Smith</td>
</tr>
</tbody>
</table>

**Relational Port Configuration**

On the Ports panel, define groups of output ports and map the nodes from the XML output schema to the ports.

If the output XML schema has more than one element that can be a root element, you can choose a node from the XML output schema to be the root element. To define an a node as a root, click **Choose Hierarchy**. The Developer tool displays only the nodes from the root level and below the root level in the **Hierarchical Output** area.

Create ports using one of the following methods:

**Drag nodes to ports**

Drag nodes from the **Hierarchical Output** area to the **Relational Ports** area. If you drag a node to a group, the Developer tool adds a port to the group. Otherwise, it creates a group with the port in it.

**Manually create the ports**

To create a port, select an empty field in the **Relational Ports** area and click **New > Field**. If you do not select a field inside a group, the Developer tool creates a group and adds the port to the group.

When you drag nodes to the **Relational Ports** area, the Developer tool updates the location field with the location of the node in the XML hierarchy. If you manually create ports, you must map a node to the port. Click the **Location** column and select a node from the list.
When you drag a multiple-occurring node into a group that contains the parent element, you can configure the number of child element occurrences to include. Or, you can replace the parent group with the multiple-occurring child group in the transformation output.

To create a group, drag a node to an empty output column in the **Relational Ports** area. If you drag a multiple-occurring child node to an empty output column, the Developer tool asks you to relate the group to other output groups. When you select a group, the Developer tool creates keys to relate the groups.

You can also create a new group by clicking **New > Group** in the **Relational Ports** area. Enter a name for the group.

Configure related groups of output ports in the **Relational Ports** area. When the Developer tool prompts you to relate output groups, it adds the keys to the groups. You can also manually add ports to represent keys.

Click **Show As Hierarchy** to display the output ports in a hierarchy. Each child group appears underneath the parent group.

To view lines that connect the ports with the XML nodes, click **Show Lines**. Select to view all the connection lines or just the lines for selected ports.

**Pass-Through Ports**

You can configure pass-through ports in a Data Processor transformation. Pass-through ports are input and output ports that receive input data and return the same data to a mapping without changing it.

You can configure pass-through ports in a Data Processor transformation instance that is in a mapping. You cannot add pass-through ports to a reusable transformation.

To add a pass-through port, drag a port from another transformation in the mapping. In the transformation, you can also click **New > Add > Port** to add a pass-through port.

**Startup Component**

A startup component defines the component that starts the processing in the Data Processor transformation. Configure the startup component on the **Overview** view.

A Data Processor transformation can contain multiple components to process data. Each component might contain other components. You must identify which component is the entry point for the transformation.

When you configure the startup component in a Data Processor transformation, you can choose an XMap or a script component as the startup component. You can select one of the following types of components:

- **Parser.** Converts source documents to XML. The output of a parser is always XML. The input can have any format, such as text, HTML, Word, PDF, or HL7.
- **Mapper.** Converts an XML source document to another XML structure or schema.
- **Serializer.** Converts an XML file to an output document of any format. The output of a serializer can be any format, such as a text document, an HTML document, or a PDF.
- **XMap.** Transforms an input XML document into an output XML document of another structure. You can transform the same XML documents as in a Mapper.
- **Streamer.** Splits large input documents, such as multi-gigabyte data streams, into segments. The streamer processes documents that have multiple messages or records in them, such as HIPAA or EDI files.
- **Transformer.** Modifies the data in any format. Adds, removes, converts, or changes text. Use Transformers with a parser, Mapper, or Serializer. You can also run a transformer as stand-alone component.
Note: If the startup component is not an XMap, you can also configure the startup component in a script instead of in the Overview view.

Schema References

The Data Processor transformation requires an XML schema to define the input or output XML hierarchy for each component that you define in the transformation. Add schema references in the transformation References view.

The Data Processor transformation references schema objects in the Model repository. The schema objects can exist in the repository before you create the transformation. You can also import schemas from the transformation References view.

The schema encoding should match the input encoding for Serializer or Mapper objects. The schema encoding should match the output encoding for Parser objects. Configure the working encoding in the transformation Settings view.

A schema can reference additional schemas. The Developer tool shows the namespace and prefix for each schema that the Data Processor transformation references. When you reference multiple schemas with empty namespaces the transformation is not valid.

Data Processor Transformation Settings

Configure code pages, XML processing options, and logging settings in the Data Processor transformation Settings view.

Character Encoding

A character encoding is a mapping of the characters from a language or group of languages to hexadecimal code.

When you design a script, you define the encoding of the input documents and the encoding of the output documents. Define the working encoding to define how the IntelliScript editor displays characters and how the Data Processor transformation processes the characters.

Working Encoding

The working encoding is the code page for the data in memory and the code page for the data that appears in the user interface and work files. You must select a working encoding that is compatible with the encoding of the schemas that you reference in the Data Processor transformation.
The following table shows the working encoding settings:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the Data Processor Default Code Page</td>
<td>Uses the default encoding from the Data Processor transformation.</td>
</tr>
<tr>
<td>Other</td>
<td>Select the encoding from the list.</td>
</tr>
</tbody>
</table>
| XML Special Characters Encoding | Determines the representation of XML special characters. You can select None or XML.  
- None.  
  - Entity references for XML special characters are interpreted as text. For example, the character &gt; appears as &gt;  
  Default is none.  
- XML. Convert to & < > ' '  
  Entity references for XML special characters are interpreted as regular characters. For example, &gt; appears as the following character: &gt; |

### Input Encoding

The input encoding determines how character data is encoded in input documents. You can configure the encoding for additional input ports in a script.

The following table describes the encoding settings in the Input area:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
</table>
| Use Encoding Specified in Input Document | Use the codepage that the source document defines, such as the encoding attribute of an XML document.  
If the source document does not have an encoding specification, the Data Processor transformation uses the encoding settings from the Settings view. |
| Use Working Encoding           | Use the same encoding as the working encoding.                              |
| Other                          | Select the input encoding from a drop-down list.                           |
XML Special Characters Encoding

Determines the representation of XML special characters. You can select **None** or **XML**.

- **None**.
  - Leave as &amp; &lt; &quot; &apos;
  - Entity references for XML special characters are interpreted as text, for example, the character &gt; appears as &gt;
  - Default in None.
- **XML**. Convert to &amp; &lt; &quot; &apos;
  - Entity references for XML special characters are interpreted as regular characters. For example, &gt; appears as the following character:
  

Byte Order

Describes how multi-byte characters appear in the input document. You can select the following options:

- **Little-endian**. The least significant byte appears first. Default.
- **Big-endian**. The most significant byte appears first.
- **No binary conversion**.

Output Encoding

The output encoding determines how character data is encoded in the main output document.

The following table describes the encoding settings in the **Output** area:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Working Encoding</td>
<td>The output encoding is the same as the working encoding.</td>
</tr>
<tr>
<td>Other</td>
<td>The user selects the output encoding from the list.</td>
</tr>
<tr>
<td>XML Special Characters Encoding</td>
<td>Determines the representation of XML special characters. You can select None or XML.</td>
</tr>
</tbody>
</table>
|                              | - **None**.
|                              |  - Leave as &amp; &lt; &quot; &apos;
|                              |  - Entity references for XML special characters are interpreted as text, for example, the character &gt; appears as &gt;
|                              |  - Default.
|                              | - **XML**. Convert to &amp; &lt; &quot; &apos;
|                              |  - Entity references for XML special characters are interpreted as regular characters. For example, &gt; appears as the following character:
|                              | >                                                                          |
**Rules and Guidelines for Character Encoding**

Use the following rules and guidelines when you configure encodings:

- To increase performance, set the working encoding to be the same encoding as the output document.
- Set the input encoding to the same encoding as the input document.
- Set the output encoding to the same encoding as the output document.
- For languages that have multiple-byte characters, set the working encoding to UTF-8. For the input and output encoding, you can use a Unicode encoding such as UTF-8 or a double-byte code page such as Big5 or Shift_JIS.

**Output Settings**

Configure output control settings to control whether the Data Processor transformation creates event logs and saves output documents.

You can control the types of messages that the Data Processor transformation writes to the design-time event log. If you save the parsed input documents with the event logs, you can view the context where the error occurred in the Event view.

The following table describes the settings in the Design-Time Events area:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
</table>
| Log design-time events   | Determines whether to create a design-time event log. By default, the Data Processor transformation logs notifications, warnings, and failures in the design-time event log. You can exclude the following event types:  
- Notifications  
- Warnings  
- Failures |
| Save parsed documents    | Determines when the Data Processor transformation saves a parsed input document. You can select the following options:  
- Always  
- Never  
- On failure  
The default is always. |
The following table describes the settings in the Run-Time Events area:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
</table>
| Log run-time events  | Determines whether an event log is created when you run the transformation from a mapping.  
- Never.  
- On failure  
The default is Never. |

The following table describes the settings in the Output area:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
</table>
| Disable automatic output | Determines whether the Data Processor transformation writes the output to the standard output file. Disable standard output in the following situations:  
- You pass the output of a Parser to the input of another component before the transformation creates an output file.  
- You use a WriteValue action to write data directly to the output from a script instead of passing data through the output ports. |
| Disable value compression| Determines whether the Data Processor transformation uses value compression to optimize memory use.  
Important: Do not disable value compression except when Informatica Global Customer Support advises you to do disable it. |

Processing Settings

The processing settings define how the Data Processor transformation processes an element without a defined datatype. The settings affect scripts. The settings do not affect elements that an XMap processes.

The following table describes the processing settings that affect XML processing in scripts:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treat as xs:string</td>
<td>The Data Processor transformation treats an element with no type as a string. In the Choose XPath dialog box, the element or attribute appears as a single node.</td>
</tr>
<tr>
<td>Treat as xs:anyType</td>
<td>The Data Processor transformation treats an element with no type as anyType. In the Choose XPath dialog box, the element or attribute appears as a tree of nodes. One node is of xs:string type, and all named complex data types appear as tree nodes.</td>
</tr>
</tbody>
</table>

XML Output Configuration

The XML generation settings define characteristics of XML output documents.
The following table describes the XML generation settings in the **Schema Title** area:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema location</td>
<td>Defines the schemaLocation for the root element of the main output document.</td>
</tr>
<tr>
<td>No namespace schema location</td>
<td>Defines the xsi:noNamespaceSchemaLocation attribute of the root element of the main output document.</td>
</tr>
</tbody>
</table>

Configure XML Output Mode settings to determine how the Data Processor transformation handles missing elements or attributes in the input XML document. The following table describes the XML generation settings in the **XML Output Mode** area:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>As is</td>
<td>Do not add or remove empty elements. Default is enabled.</td>
</tr>
<tr>
<td>Full</td>
<td>All required and optional elements defined in the output schema are written to the output. Elements that have no content are written as empty elements.</td>
</tr>
<tr>
<td>Compact</td>
<td>Removes empty elements from the output.</td>
</tr>
<tr>
<td></td>
<td>If <strong>Add for Elements</strong> is enabled, then the Data Processor transformation removes only the optional elements.</td>
</tr>
<tr>
<td></td>
<td>If <strong>Add for Elements</strong> is disabled, the Data Processor transformation removes all empty elements. The XML output might not be valid.</td>
</tr>
</tbody>
</table>

The following table describes the XML generation settings in the **Default Values for Required Nodes** area:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add for elements</td>
<td>When the output schema defines a default value for a required element, the output includes the element with a default value. Default is enabled.</td>
</tr>
<tr>
<td>Add for attributes</td>
<td>When the output schema defines a default value for a required attribute, the output includes the attribute with its default value. Default is enabled.</td>
</tr>
<tr>
<td>Validate added values</td>
<td>Determines whether the Data Processor transformation validates empty elements that are added by the Full mode output. Default is disabled.</td>
</tr>
<tr>
<td></td>
<td>If <strong>Validate added values</strong> is enabled and the schema does not allow empty elements, the XML output might not be valid.</td>
</tr>
</tbody>
</table>
The following table describes the XML generation settings in the Processing Instructions area:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add XML processing instructions</td>
<td>Defines the character encoding and XML version of the output document. Default is selected.</td>
</tr>
<tr>
<td>XML version</td>
<td>Defines the XML version. The XML version setting has the following options:</td>
</tr>
<tr>
<td></td>
<td>- 1.0</td>
</tr>
<tr>
<td></td>
<td>- 1.1</td>
</tr>
<tr>
<td></td>
<td>Default is 1.0.</td>
</tr>
<tr>
<td>Encoding</td>
<td>Defines the character encoding that is specified in the processing instruction. The Encoding setting has the following options:</td>
</tr>
<tr>
<td></td>
<td>- Same as output encoding. The output encoding in the processing instruction is the same as the output encoding defined in the Data Processor transformation settings.</td>
</tr>
<tr>
<td></td>
<td>- Custom. Defines the output encoding in the processing instruction. The user types the value in the field.</td>
</tr>
<tr>
<td>Add custom processing instructions</td>
<td>Adds other processing instructions to the output document. Enter the processing instruction exactly as it appears in the output document. Default is Disabled.</td>
</tr>
</tbody>
</table>

The following table describes the XML generation settings in the XML Root area:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add XML root element</td>
<td>Adds a root element to the output document. Use this option when the output document contains more than one occurrence of the root element defined in the output schema. Default is Disabled.</td>
</tr>
<tr>
<td>Root element name</td>
<td>Defines a name for the root element to add to the output document.</td>
</tr>
</tbody>
</table>

Events

An event is a record of a processing step from a component in the Data Processor transformation. In a script, each anchor, action, or transformer generates an event. In an XMap, each mapping statement generates an event.

You can view events in the Data Processor Events view.

Event Types

The Data Processor transformation writes events in log files. Each event has an event type that indicates if the event was successful, the event failed, or if the event ran with errors.

A component can generate one or more events. The component can pass or fail depending on whether the events succeed or fail. If one event fails, a component fails.
The following table describes the types of events that the Data Processor transformation generates:

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notification</td>
<td>Normal operation.</td>
</tr>
<tr>
<td>Warning</td>
<td>The Data Processor transformation ran, but an unexpected condition occurred. For example, the Data Processor transformation wrote data to the same element multiple times. Each time the element is overwritten, the Data Processor transformation generates a warning.</td>
</tr>
<tr>
<td>Failure</td>
<td>The Data Processor transformation ran, but a component failed. For example, a required input element was empty.</td>
</tr>
<tr>
<td>Optional Failure</td>
<td>The Data Processor transformation ran, but an optional component failed. For example, an optional anchor was missing from the source document.</td>
</tr>
<tr>
<td>Fatal Error</td>
<td>The Data Processor transformation failed because of a serious error. For example, the input document did not exist.</td>
</tr>
</tbody>
</table>

**Data Processor Events View**

The Data Processor Events view displays events when you run a Data Processor transformation from the Developer tool.

The Data Processor Events view has a Navigation panel and a Details panel. The Navigation panel contains a navigation tree. The navigation tree lists the components that the transformation ran in chronological order. Each node has an icon that represents the most serious event below it in the tree. When you select a node in the Navigation panel, events appear in the Details panel.

The navigation tree contains the following top-level nodes:
- Service Initialization. Describes the files and the variables that the Data Processor transformation initializes.
- Execution. Lists the components that the Script or XMap ran.
- Summary. Displays statistics about the processing.

When you run an XMap, each node name in the navigation panel has a number in square brackets, such as [5]. To identify the statement that generated the events for the node, right-click in the statements grid and select Go to Row Number. Enter the node number.

When you run a Script and double-click an event in the Navigation panel or the Details panel, the Script editor highlights the script component that generated the event. The Input panel of the Data Viewer view highlights the part of the example source document that generated the event.

**Logs**

A log contains a record of the Data Processor transformation. The Data Processor transformation writes events to logs.

The Data Processor transformation creates the following types of logs:
Design-time event log

The design-time event log contains events that occur when you run the Data Processor transformation in the Data Viewer view. View the design-time log in the Events view.

Run-time event log

The run-time event log contains events that occur when you run the Data Processor transformation in a mapping. You can view the run-time event log in a text editor or you can drag a run-time event log into the Events view of the Data Processor transformation.

User log

The user log contains events that you configure for components in a script. The Data Processor transformation writes to the user log when you run it from the Data Viewer view and when you run it in a mapping. You can view the user log in a text editor.

Design-Time Event Log

The design-time event log contains the events that occur when you run the Data Processor transformation from the Data Viewer in the Developer tool.

When you run a Data Processor transformation from the Data Viewer view, the design-time event log appears in the Data Processor Events view. By default, the design-time event log contains notifications, warnings, and failures. In the transformation settings, you can configure the Data Processor transformation to exclude one or more event types from the log.

When you save the input documents with the log, you can click an event in the Data Processor Events view to find the location in the input document that generated the event. When you configure the Data Processor transformation settings, you can choose to save the input files for every run or only on failure.

The design-time event log is named events.cme. You can find the design-time event log for the last run of the Data Processor transformation in the following directory:

`C:\<Installation_directory>\clients\DT\CMReports\Init\events.cme`

The Data Processor transformation overwrites the design-time event log every time you run the transformation in the Data Viewer. Rename the design-time event log if you want to view it after a later run of the transformation, or if you want to compare the logs of different runs. When you close the Developer tool, the Developer does not save any files in the Design-Time Event Log.

Run-Time Event Log

The run-time event log records the events that occur when you run the Data Processor transformation in a mapping.

If the Data Processor transformation completes the run with no failures, it does not write an event log. If there are failures in the run, Data Processor transformation runs a second time and writes an event log during the second run. The run-time event log is named events.cme.

On a Windows machine, the run-time event log in in the following directory:

`C:\<Installation_Directory>\clients\DT\CMReports\Tmp\`

On a Linux or UNIX machine, the run-time event log for a root user in the following directory:

`/root/<Installation_Directory>/clients/DT/CMReports/Tmp`

On a Linux or UNIX machine, you can find the run-time event log for a non-root user in the following directory:

`/home/[UserName]<Installation_Directory>/DT/CMReports/Tmp`

Use the configuration editor to change the location of the run-time event log.
Viewing an Event Log in the Data Processor Events View

Use the **Data Processor Events** view to view a design-time event log or a run-time event log.

Open Windows Explorer, and then browse to the event log file you want to view. Drag the log from the Windows Explorer window to the **Data Processor Events** view. Right-click in the **Data Processor Events** view, and then select **Find** to search the log.

**Note:** To reload the most recent design-time event log, right-click the **Data Processor Events** view, and then select **Reload Project Events**.

User Log

The user log contains custom messages that you configure about failures of components in a script.

The Data Processor transformation writes messages to the user log when you run a script from the **Data Viewer** view and when you run it in a mapping.

When a script component has the **on_fail** property, you can configure it to write a message to the user log when it fails. In the script, set the **on_fail** property to one of the following values:

- LogInfo
- LogWarning
- LogError

Each run of the script produces a new user log. The user log file name contains the transformation name with a unique GUID:

```
<Transformation_Name>_GUID.log
```

For example, `CalculateValue_Aa93a9d14-a01f-442a-b9cb-c9ba5541b538.log`

On a Windows machine, you can find the user log in the following directory:

```
c:\Users\[UserName]\AppData\Roaming\Informatica\DataTransformation\UserLogs
```

On a Linux or UNIX machine, you can find the user log for the root user in the following directory:

```
/Installation_Directory\DataTransformation/UserLogs
```

On a Linux or UNIX machine, you can find the user log for a non-root user in the following directory:

```
homed/Installation_Directory\DataTransformation/UserLogs
```

Data Processor Transformation Development

To create a Data Processor transformation, create the transformation, choose the XML schemas that define the input or output, and define XMap or Script objects in the transformation.

Create the Data Processor Transformation

Create the Data Processor transformation in the Model Repository.

1. In the Developer tool, click **File > New > Transformation**.
2. Select the Data Processor transformation and click **Next**.
3. Enter a name for the transformation and browse for a Model repository location to put the transformation.
4. Click Finish.
   The Developer tool creates the transformation in the repository. The Overview view appears in the Developer tool. You can update the transformation ports after you define the schemas and objects in the transformation.

5. Optionally, click the References view to choose the schema objects that define input or output hierarchies for each component that you plan to create.

Select the Schema Objects

Choose the schema objects that define the input or output XML hierarchies for each XMap or script component that you plan to create.

You can add schema references on the References view or you can add the schema references when you create Script or XMap objects. A schema object must exist in the Model repository before you can reference it in a Script or XMap.

1. In the Data Processor transformation References view, click Add.
2. If the schema object exists in the Model repository, browse for and select the schema.
3. If the schema does not exist in the Model repository, click Create a new schema object and import a schema object from an .xsd file.
4. Click Finish to add the schema reference to the Data Processor transformation.

Step 3. Create Objects

Create a script or XMap object on the Data Processor transformation Objects view. After you create the object, you can open the object from the Objects view in order to configure it.

Creating a Script

Create a Script object and define the type of Script component to create. Optionally, you can define a script reference and example source file.

1. In the Data Processor transformation Objects view, click New.
2. Enter a name for the script object and click Next.
3. Choose to create a Parser or Serializer. Select Other to create a Mapper, Transformer, or Streamer component.
4. Enter a name for the component.
5. If the component is the first component to process data in the transformation, enable Set as startup component.
6. Click Next if you want to enter a schema reference for this script. Click Finish if you do not want to enter the schema reference.
7. If you choose to create a schema reference, select Add reference to a Schema Object and browse for the Schema object in the Model repository. Click Create a new schema object to create the Schema object in the Model repository.
8. Click Next to enter an example source reference or to enter example text. Click Finish if you do not want to define an example source.
   Use a example source to define sample data and to test the Script.
9. If you choose to select an example source, select File and browse for the sample file.
   You can also enter sample text in the Text area. The Developer tool adds the text to a script.
10. Click **Finish**.
    The Script view appears in the Developer tool editor.

**Creating an XMap**

Create an XMap on the Data Transformation **Objects** view. When you create an XMap, you must have a schema that describes the input and the output XML documents. You select the element in the schema that is the root element for the input XML.

1. In the Data Processor transformation **Objects** view, click **New**.
2. Select XMap and click **Next**.
3. Enter a name for the XMap.
4. If the XMap component is the first component to process data in the transformation, enable **Set as startup component**. Click **Next**.
5. If you choose to create a schema reference, select **Add reference to a Schema Object** and browse for the Schema object in the Model repository. Click **Create a new schema object** to import a new Schema object.
6. If you have a sample XML file that you can use to test the XMap with, browse for and select the file from the file system.
    You can change the sample XML file.
7. Choose the root for the input hierarchy.
    In the **Root Element Selection** dialog box, select the element in the schema that is the root element for the input XML. You can search for an element in the schema. You can use pattern searching. Enter `*<string>` to match any number of characters in the string. Enter `?<character>` to match a single character.
8. Click **Finish**.
    The Developer tool creates a view for each XMap that you create. Click the view to configure the mapping.

**Adding an Example Source**

Choose the example source to test the Script, XMap or Library that you plan to create.

You can add an example source when you create a Script, XMap or Library. A example source must exist in the Model repository before you can reference it in a Script, XMap or Library.

You can change the sample file.

**Step 4. Create the Ports**

Configure the input and output ports in the **Overview** view.

When you configure additional input or output ports in a script, the Developer tool adds additional input ports and additional output ports to the transformation by default. You do not add input ports on the **Overview** view.

1. If you want to return rows of output data instead of XML, enable **Relational Output**.
   When you enable relational output, the Developer tool removes the default output port.
2. Select the input port datatype, port type, precision and scale.
3. If you are not defining relational output ports, define the output port datatype, port type, precision, and scale.
4. If a Script has additional input ports, you can define the location of the example input file for the ports. Click the **Open** button in the **Input Location** field to browse for the file.

5. If you enabled relational output, click **Output Mapping** to create the output ports.

6. On the Ports view, map nodes from the **Hierarchical Output** area to fields in the **Relational Ports** area.

**Step 5. Testing the Transformation**

Test the Data Processor transformation in the **Data Viewer** view.

Before you test the transformation, verify that you defined the startup component. You can define the startup component in a script or you can select the startup component on the **Overview** tab. You also need to have chosen an example input file to test with.

1. Open the **Data Viewer** view.
2. Click **Run**.
   
   The Developer tool validates the transformation. If there is no error, the Developer tool shows the example file in the **Input** area. The output results appear in the **Output** panel.
3. Click **Show Events** to show the **Data Processor Events** view.
4. Double-click an event in the **Data Processor Events** view in order to debug the event in the **Script editor**.
5. Click **Synchronize with Editor** to change the input file when you are testing multiple components, each with a different example input file.
   
   If you modify the example file contents in the file system, the changes appear in the **Input** area.

**Data Processor Transformation Import and Export**

You can export a Data Processor transformation as service and run it from a Data Transformation repository. You can also import a Data Transformation service to the Developer tool. When you import a Data Transformation service, the Developer tool creates a Data Processor transformation from the service.

**Note:** When you import a Data Transformation service to the Model repository, the Developer tool imports the associated XML schemas to the repository. If you modify the schema in the repository, sometimes the changes do not appear in the transformation schema references. You can close and open the Model repository connection, or close and open the Developer tool to cause the schema changes to appear in the transformation.

**Exporting the Data Processor Transformation as a Service**

You can export the Data Processor transformation as a Data Transformation service. Export the service to the file system repository of the machine where you want to run the service. You can run the service with PowerCenter, user-defined applications, or the Data Transformation CM_console command.

1. In the **Object Explorer** view, right-click the Data Processor transformation you want to export, and select **Export**.
   
   The **Export** dialog box appears.
2. Select **Informatica > Export Data Processor Transformation** and click **Next**.
   
   The **Select** page appears.
3. Click **Next**.
The Select Service Name and Destination Folder page appears.

4. Choose a destination folder:
   - To export the service on the machine that hosts the Developer tool, click Service Folder.
   - To deploy the service on another machine, click Folder. Browse to the \ServiceDB directory on the machine where you want to deploy the service.

5. Click Finish.

Importing a Data Transformation Service

You can import a Data Transformation service .cmw file to the Model repository and create a Data Processor transformation.

1. Click File > Import,
   The Import dialog box appears.
2. Select Informatica Import Data Transformation Service and click Next.
   The Import Data Transformation Service page appears.
3. Browse to the .cmw file that you want to import.
   The Developer tool imports the schemas and example data with the .cmw file. The Developer tool creates a transformation by default.
4. Double-click the transformation in the Navigator to edit it.
This chapter includes the following topics:

- XMap Overview, 25
- XMap Schemas, 26
- Mapping Statements, 27
- Creating Mapping Statements, 31
- XPath Expressions, 32
- XMap Variables, 36
- XMap Example, 37

**XMap Overview**

The XMap object transforms an XML source document to another XML document with a different hierarchy structure. You can use XPath statements to define a range of conditions to transform the data. Use a grid in the Developer tool to define and manage mapping statements.

For example, you might receive customer invoices in an XML document which contains itemized orders sorted by customer name. You want sort order data by time period in an output XML document. The output XML document must sort customers who ordered within specified time periods by customer name, and include their contact information and order totals.

In a Data Processor, you configure an XMap object to map the data in the input XML document to the data in the output XML document. Use XPath expressions to define time periods, find relevant customer order information, and sort the data. You can also use XPath expressions to compute order totals.

When you configure an XMap object, you map nodes from the input XML schema to nodes in the output XML schema. You link input nodes to output nodes to create mapping statements. You can edit the mapping statements in the XMap editor. You can define conditions to transform the data and you can define predicates to filter the data in the mapping statements.

When you open an XMap object, the XMap editor appears:
The XMap editor shows you the input XML schema and the output XML schema. You can link input and output nodes in this view. Click and drag nodes from the input schema to elements in the output schema to create a map. When you drag an input node to an output node, the Developer tool creates a mapping statement in the grid below.

Use the grid to create simple or detailed mapping definitions. The mapping definition describes how to map input elements to output elements. You can drag elements from the input or output schemas above into fields in the grid below to include them in mapping statements.

You can also check the nodes that a mapping statement references. When you click a mapping statement in the grid, the XMap editor highlights the nodes in the schemas.

You can add XPath expressions which locate and process node data. You can also use the XPath editor to text expressions.

**XMap Schemas**

The top area of the XMap editor shows the input and output schemas. You cannot change the schemas in the XMap editor, but you can change the input and output root, add variables, and customize the schema view.

You can select the following schema options in the XMap editor:

**Customize view**

Changes the way the schema appears. Includes nodes for sequence, all and choice order indicators. View these nodes to understand the schema logic. These nodes do not affect the mapping.

**Search**

Search for elements in the schema. You can search in the input and output schemas separately.

**Variables**

Define variables to store data. You can map nodes to variables and map variables to nodes. You can also map variables to variables. When you create a variable, the variable appears at the bottom of both schemas.
Select Input or Output Root

Change the root element in the input or output schema. You might change the root element in order to reference a different part of the schema. For example, you might select a root element to match the root of an input or output XML document.

Choose the example source

Define the example source file. The example source file is in the Model repository. Use an example source to test the transformation and to test XPath expressions.

Mapping Statements

A mapping statement describes how to map nodes from the input schema to the output schema in an XMap. When you drag a node from the input schema to the output schema, the Developer tool creates mapping statements in the XMap editor. You can edit the XMap statements to add predicates, XPath functions, and references to variables. You can define conditions for mapping the elements.

An input element XPath expression determines when the Data Processor transformation performs the mapping statement. The Data Processor transformation performs the mapping statement for each occurrence of the input element or expression.

A mapping statement might be a parent to a group of child statements. Child statements appear indented from the parent in the XMap editor. When the Data Processor transformation performs the parent statement, it performs the child statements in the group.

Each mapping statement has a row number.

Mapping Statement Types

Mapping statement types define XMap mapping logic. Define the mapping statement type based on whether you want to map a simple input value to an output value, iterate over an element, or perform the mapping based on a condition.

Create a statement by dragging an input schema element to an output schema element, or adding a mapping statement to the grid. When you create a statement, the Data Processor transformation identifies a mapping statement type based on whether the element is a simple element, a complex element, or a repeating element.

The basic mapping statement type is a Map, which maps a simple input value to a simple output value. Other mapping statements identify conditions or alternatives for mapping logic, or group a set of logical statements.

You can define the following types of mapping statements in the grid:

Map
Maps a simple input element to a simple output element. A Map statement is the basic building block of the XMap.

Group
A logical group of statements. Other mapping statement types are nested under the Group statement.

Repeating Group
A group statement that the Data Processor transformation performs each time the input element appears in the input document. The Repeating Group contains Map statements which are iterated. The Repeating Group identifies the element used to iterate the group.
Router

Contains a group of Option statements, and selects only the Option statement whose condition criteria matches the input. If none of the Options apply, a Default action is taken, if there is a Default statement. If none of the Options apply and there is no Default statement, the Router fails.

Option

One or more Option statements are nested under the Router statement. The Option statement is like a Group statement, and contains a logical group of statements. The Option statement defines a condition to map the input element to the output element.

Default

One Default statement can be nested under the Router statement. The Default statement is performed when none of the Option statements apply. If the all the Option statements fail and there is no Default statement, the Router fails.

Run XMap

Calls another XMap object in the Data Processor transformation.

Mapping statements contain fields that you can configure to customize the statement. You can configure the input, output, and condition for mapping an input element to an output element.

Configure whether to skip a mapping statement when it fails or there is no input. Configure whether the Data Processor transformation adds an output element or matches an existing element with a value from a mapping statement.

Mapping Statement Fields

Mapping statements contain fields that you can configure to customize the statement. You can configure the input, output, and condition for mapping the element. Configure whether to skip a statement when it fails or has missing input. Configure whether the Data Processor transformation adds a new output element or matches an existing element with a value from a mapping statement.

You can configure the following mapping fields:

Name

A name for the statement. You can change the name at any time. The name identifies statements so you can find them in the mapping grid or in an event log. Statement names do not have to be unique.

Statement Type

Identifies the statement operation. If the statement calls another XMap, the Statement Type field contains the name of the XMap to call.

Input

An XPath expression that evaluates to a sequence of nodes or values. The mapping statement type determines how the Data Processor transformation uses the nodes or values in the mapping.

Condition

An XPath expression that defines a condition for mapping the element. A condition is similar to a predicate expression in the Input column. If you define an Input XPath expression and a Condition XPath expression for the same mapping statement, the Data Processor transformation applies the Condition XPath to the result of the Input XPath.

Skip Missing Input

When enabled, and the input element is not in the XML document, the Data Processor transformation skips the statement without error. When Skip Missing Input is disabled, the statement fails when the element is not in the input XML document.
Default

The default value to use when an element is missing from the input. For example, you can define a default value to initialize a counter.

On Fail

If the statement fails, you can choose how to handle the failure. Enable Skip or Skip Iteration if it is okay to skip the statement because it failed. Skip iteration appears when the statement is a Repeating Group statement. Select Propagate to force the parent statement to fail too.

Output

An XPath expression that defines the value of the element in the output XML based on the results of the Input XPath expression. If the mapping statement has child statements, the Output field provides the context for the child statements.

Mode

Determines whether the Data Processor transformation adds an output node or matches an existing node with a value from a mapping statement. Choose one of the following modes:

Add

Creates a new element in the output XML document. If the element is not multiple-occurring, and the same values exist in the output, the mapping statement fails.

Match

Select Match when you expect to always find a match for the value in the output elements. The mapping statement fails if the Data Processor transformation finds no match for the value in the output XML document. Each element value must already exist in the output.

Match or Add

Select Match or Add when you expect to match a node value or create a new node if it does not exist in the output XML document.

Group Statements

A Group statement defines a logical group of statements. A Group statement is a parent of the mapping statements nested below it in the grid. The group node defines the context for nodes that you reference in the child statements.

Each statement in a group is related by context. The input or output of the Group statement defines the context. The input to each statement in a group might be a descendent of the same node in the input schema.

Group Statement Example

A schema has a single occurring Employee node. Employee has FirstName and LastName child nodes:

```
Employee
  FirstName
  LastName
```

If you create a group mapping statement and configure Employee as the input, each mapping statement that you include in the group is within the context of Employee.

In the following figure, statement 1 is the Group statement:
The Input column for the Group statement shows that the input is the parent node of FirstName and LastName. Statement 2 and statement 3 are child statements of statement 1. The child statements appear indented from the parent statement. For each input Employee element, map the FirstName element and the LastName element to the output.

**Repeating Group Statements**

A Repeating Group statement is a group statement that can occur multiple times. The input is an XPath expression that can evaluate to a sequence of nodes or values.

The Data Processor transformation performs the Repeating Group statement for each node or value that is a result of the Input XPath expression.

**Repeating Group Statement Example**

You might have an input schema with the following hierarchy:

```
Employees
  Employee (Unbounded)
    LastName
    FirstName
```

When you drag Employee to an output node in the XMap Editor, the Developer tool creates a Repeating Group mapping statement by default. A repeating group might contain mapping statements to return LastName and FirstName for each Employee in the input XML document.

**Router, Option, and Default Statements**

A Router statement provides more than one alternative for the mapping logic based on conditions in the input document.

When you configure a Router statement, you must configure at least one Option statement under it. The Data Processor transformation evaluates the input and condition fields in the Option statement.

The Data Processor transformation accepts an Option statement when the results of the Input XPath expression and a Condition XPath expression evaluate to a singleton. A singleton is a sequence of a single node value. If you do not define an Input XPath value, the Data Processor transformation accepts the Option statement when the Condition XPath is true.

You can configure multiple Option statements in the same Router group. The Data Processor transformation performs the first Option statement that it accepts. It does not perform any Option statement below it in the group. If the Data Processor transformation does not accept an Option statement, it tests the next Option statement.

When the transformation accepts no statements, the Router statement can fail. If the Option statement has a condition that is true but the mapping statements inside it fail and propagate the failure, the Router fails.

You can configure the mapping to skip the Router if the Router might fail.

The Option statement can contain one or more child statements of any type, including Map, Group, Repeating Group, Run XMap, and other Router statements.

For example, an Option statement might contain the following condition:

```
EmployeeID="100"
```
When the EmployeeID is 100, the condition is true. The child statement in the grid defines the mapping statement to evaluate when the condition is true.

**Router Statement Example**

An XMap contains a repeating group under the context of Employee. The first child statement in the group is a Router statement. The Router has one Option statement. The Option statement has a condition that checks if the value of Role is equal to the value of Manager. If the Role equals Manager, the option is true. The mapping evaluates the next statement for the option. The Data Processor transformation calls the EmployeeToWorker XMap to map elements to Manager.

If the role is not equal to manager, the Default statement is true. The mapping evaluates the next statement for the Default option. The default mapping statement calls EmployeeToWorker XMap to map the elements to Worker.

The following figure shows the Router statement with an Option statement and a Default statement:

![Figure showing router statement with option and default statements]

**Run XMap Statements**

Use a Run XMap statement to call another XMap.

When you create a Run XMap mapping statement, the Developer tool lists the XMap objects in the transformation. Select the XMap to call. The Developer tool creates a mapping statement with the XMap name in the Statement Type field.

The input and output root elements in the called XMap must be the same type as the input and output values being passed to it from the calling XMap.

**Creating Mapping Statements**

Create mapping statements in the XMap editor. You can create mapping statements by dragging nodes from the input schema to the output schema and you can define the statements in the mapping statement grid.

When you drag a node from the input schema to the output schema, the Developer tool creates a Map, Group, or Repeating Group mapping statement in the grid. You can update the mapping statement. The Input and Output fields contain the elements from the schema. If you want to create mapping statements to provide context or to define Router options, you can type statements in the grid.

When you select a mapping statement in the grid, the XMap editor highlights the nodes from the input and output schemas that are in the grid statement.

You can copy a statement from one row to another row in the grid. If the row is not valid for the location in which you copy it, the XMap editor displays a dialog box with an error that the statement is not valid the current context. You can change the input and the output XPath statements on the dialog box to adjust the mapping statement context, or you can change the input and output XPath fields in the grid.

Use the following steps to define mapping statements in the grid:

1. In the grid options, click **New**.
2. Select the type of mapping statement from the list.
If you choose Run XMap, the Developer tool shows a list of the XMap objects in the transformation.

3. Optionally, define a name for the mapping statement.

4. Drag an element from the input schema to the Input field to define the input to the mapping statement. Or, you can configure an XPath expression or a constant in the Input field.

5. Drag an element from the output schema to the Output field to define the output of the mapping statement. You can also configure an XPath expression in the Output field.

6. To create an XPath expression in an Input, Output, or Condition field, click the **Open** button in the field to open the Expression Editor.

7. To change the mapping statement type, click the **Open** button in the **Statement Type** field. Choose the mapping statement type from the list.

### XPath Expressions

Create an XPath expression to identify specific elements in XML documents, select nodes, or check for conditions in the data. Create XPath expressions in the Input, Condition, or Output field of an XMap mapping statement.

XPath is a syntax to define parts of an XML document. Use XPath to select sequences of nodes or values in an XML document. XPath includes a library of standard functions that you can use to select data.

You can define XPath 2.0 expressions in the Data Processor transformation. When you configure Output XPath expressions, you can use a subset of the XPath 2.0 syntax when you define mapping statements for Add mode or Match or Add mode.

For more information about XPath, refer to your XPath documentation.

The following table describes some XPath expressions:

<table>
<thead>
<tr>
<th>XPath Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodename</td>
<td>Selects all child nodes of the given name in the context.</td>
</tr>
<tr>
<td>. (dot)</td>
<td>Selects current node.</td>
</tr>
<tr>
<td>..</td>
<td>Selects parent of current node.</td>
</tr>
<tr>
<td>@</td>
<td>Selects attribute.</td>
</tr>
<tr>
<td>/</td>
<td>Selects from root node or child of current node if preceded by node. When the path starts with a slash (/) it represents an absolute path to an element.</td>
</tr>
<tr>
<td>//</td>
<td>Selects nodes anywhere in the document or descendants of current node if preceded by node.</td>
</tr>
</tbody>
</table>
The following table lists some XPath expressions and the result of each expression:

<table>
<thead>
<tr>
<th>XPath Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>/bookstore</td>
<td>Selects the root bookstore node.</td>
</tr>
<tr>
<td>bookstore/book</td>
<td>Selects book nodes that are children of all bookstore nodes.</td>
</tr>
<tr>
<td>//book</td>
<td>Selects the book nodes in the document in all locations.</td>
</tr>
<tr>
<td>bookstore/book</td>
<td>Selects all book nodes that are descendants of the bookstore nodes.</td>
</tr>
<tr>
<td>/bookstore/*</td>
<td>Selects all child nodes of bookstore root element.</td>
</tr>
<tr>
<td>//*</td>
<td>Returns a sequence of all elements in the document.</td>
</tr>
</tbody>
</table>

**Predicates**

A predicate is an expression that you configure to find a node in an XML document. You can configure the expression to find a specific value. Create a predicate in an Input or Condition, or Output field of a mapping statement.

When you define a predicate, enclose the expression in square brackets [ ] after the node.

`/<node>[expression]`

For example, the following expression selects the book elements that are descendants of bookstore and have a price element with a value greater than 55.00:

`/bookstore/book[price>55.00]`

The following expression selects the title elements of the book elements that are descendants of bookstore with a price element value greater than 55.00:

`/bookstore/book[price>55.00]/title`

The following expression selects the title elements that have an attribute lang with a value of "eng":

`//title[@lang="eng"]`

**Note:** The Data Processor transformation cannot accept all XPath statements when you configure a mapping statement with the Add mode or the Match or Add mode.

**XPath Expression Editor**

Create expressions in the XPath Expression Editor.

When you click the Open button in the Input, Condition, or Output field, the Expression Editor appears:
Create expressions in the **Expression** panel.

The XPath Expression Editor has a **Navigation** panel with a function library that you can use to create XPath expressions. The functions are standard for the W3C XML Path Language. The function library also includes some functions that are specific to the Data Processor transformation.

### Data Processor Functions

The Expression Editor has Data Processor functions that you can use for the Data Processor transformation.

The following table describes Data Processor XPath functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>dp:as_xml</td>
<td>Receives a node as an input and returns the node value and the value of all the children as an XML string recursively.</td>
<td>dp:as_XML(&lt;node&gt;)</td>
</tr>
<tr>
<td>dp:get_id</td>
<td>Generates a unique ID associated with a node and returns it. You can use the ID to create primary key-foreign key relationships in the data. Map the ID to a node in the XML and map it to keys in relational data.</td>
<td>dp:get_id(&lt;node&gt;)</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Syntax</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>dp:input</td>
<td>Returns the node that provides the current input context. Use the function in the Output field to refer to a node from the Input schema.</td>
<td>dp:input()</td>
</tr>
<tr>
<td>dp:lookup</td>
<td>Retrieves data from a lookup transformer that you define in a script. You can perform an inline or external file lookup.</td>
<td>dp:lookup(&lt;lookuptable-name&gt;,&lt;lookup-value&gt;, [default-value])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Lookuptable-name. The name of the lookup transformer in the script.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Lookup-value. The value to search for.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Default value. A default return value to use if the lookup fails. If the default value is missing and the lookup fails to find a matching value, then the function fails.</td>
</tr>
<tr>
<td>dp:output</td>
<td>Returns the node that provides the current output context. Use the function in the Input field to refer to a node in the output schema.</td>
<td>dp:output()</td>
</tr>
</tbody>
</table>

**XPath Expressions Example**

The following example shows XPath expressions in mapping statements:

The XMap input document is a list of children and their hobbies. The input root is Children. Child is a multiple-occurring element within Children. Each child has a Name and multiple-occurring hobbies. Name consists of First, Initial, and Last elements.

The output is a list of the classes with the number of children in each class. The output root is Classes. Classes has an attribute that contains the total number of classes. Each input Hobby element maps to a output Class element. A Map statement concatenates the First, Initial, and Last elements into the Child output element. Another Map statement counts the number of children in each class. Another statement counts the number of classes.
The XMap contains the following expressions:

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Input or Output Expression</th>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Output</td>
<td>Class[@name = dp:input()]</td>
<td>Adds a new Class element or find a Class that matches Hobby. The dp:input() is required because the expression refers to an input element.</td>
</tr>
<tr>
<td>3</td>
<td>Input</td>
<td>concat(../Name/First,' ','../Name/Initial,' ','../Name/Last)</td>
<td>Concatenates the First, Initial, and Last names and adds spaces between them.</td>
</tr>
<tr>
<td>4</td>
<td>Input</td>
<td>dp:output()/@noOfChildren + 1</td>
<td>For each Hobby that occurs, add 1 to the number of children for that class. The dp:output() function is required because the expression refers to an output element.</td>
</tr>
<tr>
<td>5</td>
<td>Input</td>
<td>count(dp:output()//Class)</td>
<td>Counts the Class elements. The dp:output() function is required because the expression refers to an output element.</td>
</tr>
</tbody>
</table>

Creating Expressions

Create XPath expressions in the Expression Editor.

1. In the XMap statement, click the Open button in the Input, Condition, or Output field.

   The Expression Editor appears.

2. Double-click elements in the Navigation panel to add them to an expression.

3. Click Validate to validate the expression.

4. If the expression is for the Input field, click Test Expression to test the expression against the example data.

   Results appear each time the Developer tool evaluates the expression using the example data. The XPath expression can return a sequence of zero or more nodes or values. The Sequence Length indicates how many nodes the XPath expression returns.

XMap Variables

You can add variables in the XMap editor. You can map values to variables and use the variables in predicates or as temporary holders for values. You can map variables to output elements.

When you create a variable, the variable appears in the input and the output schemas in the XMap view. The Developer tool adds a dollar sign ($) to the variable name to indicate that it is a variable.

You can create a variable that is a list of multiple values. You can use a list variable for the same purpose as a multiple-occurring schema element. Configure a list variable as input for a repeating group or configure a predicate to search for a value in the list variable.

For example, you have an XML document that contains addresses. You need to create a list of all the countries from the addresses. Map the country element into a $countries variable that you define as a list.
Creating a Variable in the XMap Editor

You can create variables in the XMap editor.

1. Click Variables above the input or output schema in the XMap editor.
   The Variables dialog box appears.
2. To create a variable, click New.
3. Enter a variable name and a datatype.
4. Enable List to create a multiple-occurring variable.

XMap Example

An XML document contains employee data that includes the employee role in the company. You need to create an XML document that has the managers and employees in separate groups. You create two XMap objects in the Data Processor transformation to restructure the XML document.

The startup component is an XMap object that contains a Router statement. An Option statement checks if the Employee role is "Manager." If the role is manager, the XMap maps the employee elements to a manager output group. Otherwise, the XMap maps the employee elements to a worker group in the output XML.

The startup component XMap calls another XMap to map the Employee elements to output elements.

XML Input Data

The following text shows sample data from the input XML document:

```xml
<?xml version="1.0" encoding="windows-1252"?>
<Input xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="company
Company.xsd" xmlns="company">
  <Company>
    <Name>Hypostores</Name>
    <Department>
      <Name>Customer Service</Name>
      <Employee id="25721195">
        <FirstName>Blair</FirstName>
        <LastName>Conner</LastName>
        <Role>Manager</Role>
        <StartDate>1993-04-21</StartDate>
      </Employee>
      <Employee id="238036220">
        <FirstName>Karina</FirstName>
        <LastName>Rasmussen</LastName>
        <Role>Worker</Role>
        <StartDate>1993-08-15</StartDate>
      </Employee>
    </Department>
    <Department>
      <Name>Research and Development</Name>
      <Employee id="259089785">
        <FirstName>Thaddeus</FirstName>
        <LastName>Burt</LastName>
        <Role>Consultant</Role>
        <StartDate>1998-02-26</StartDate>
      </Employee>
    </Department>
  </Company>
</Input>
```
The data can include multiple companies. Each company has multiple departments. Each employee in a department has a role that is either a manager or another type of worker.

Input and Output XML Hierarchies

The XMap editor shows the input hierarchy in the left area of the view and the output XML hierarchy in the right area of the view.

The following figure shows the input and output XML hierarchies:

![XMap_Example_View_Top](image)

Mapping Statements in the Example

The grid area of the XMap editor contains the statements to map the input XML elements to the output XML elements.

The following figure shows the mapping statements in the grid:
The following table describes the mapping statements in the grid:

<table>
<thead>
<tr>
<th>Row</th>
<th>Statement Name</th>
<th>Type</th>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Company</td>
<td>Repeating Group</td>
<td>Company</td>
<td>The Company statement repeats for each Company element. The statement provides a context for the rest of the statements in the grid. For each company, the Data Processor transformation evaluates the child statements.</td>
</tr>
<tr>
<td>2</td>
<td>Department</td>
<td>Repeating Group</td>
<td>Department</td>
<td>The Department statement repeats for each Department element. The statement provides a context for the rest of the statements in the grid. For each department, the Data Processor transformation evaluates the child statements.</td>
</tr>
<tr>
<td>3</td>
<td>Name to Department</td>
<td>Map</td>
<td>Name</td>
<td>Maps the name to a variable $deptName.</td>
</tr>
<tr>
<td>4</td>
<td>Match Organization</td>
<td>Repeating Group</td>
<td>Department</td>
<td>Match Organization has an output expression: $tns0:Organization[tns0:Department=deptName] The statement either finds (matches) the Organization element in the output which contains a Department child element with the value in $deptName. Or, if the Department element doesn’t exist, the Department is created.</td>
</tr>
<tr>
<td>5</td>
<td>EmployeeToWorker</td>
<td>Repeating Group</td>
<td>Employee</td>
<td>The EmployeeToWorker statement repeats for each Employee element.</td>
</tr>
<tr>
<td>6</td>
<td>Employee</td>
<td>Router</td>
<td>No input</td>
<td>The Employee statement is a Router statement. The statement has no input or output.</td>
</tr>
<tr>
<td>7</td>
<td>EmployeeToMgr</td>
<td>Option</td>
<td>No input</td>
<td>The Option statement has a condition $tns0:Role=&quot;Manager&quot;. When the Role is Manager, the statement is true. The Data Processor transformation evaluates the statements inside the option.</td>
</tr>
<tr>
<td>8</td>
<td>EmployeeToWorker</td>
<td>XMap_EmployeesToRoles</td>
<td>Employee</td>
<td>Calls the XMap to pass the Employee elements to the Manager type.</td>
</tr>
</tbody>
</table>
### EmployeeToWorker XMap

The EmployeeToWorker XMap moves elements from an employee to a worker. The XMap processes one employee.

The following figure shows the EmployeeToWorker XMap in the XMap editor:

![EmployeeToWorker XMap Diagram](image)

The following table describes the mapping statements:

<table>
<thead>
<tr>
<th>Row</th>
<th>Statement Name</th>
<th>Type</th>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EmployeeToWorker</td>
<td>Group</td>
<td>Employee</td>
<td>Group statement that provides context for the rest of the mapping statements.</td>
</tr>
<tr>
<td>2</td>
<td>FirstNametoFirstName</td>
<td>Map</td>
<td>FirstName</td>
<td>Maps the first name to the first name.</td>
</tr>
<tr>
<td>3</td>
<td>LastNametoLastName</td>
<td>Map</td>
<td>LastName</td>
<td>Maps the last name to the last name.</td>
</tr>
<tr>
<td>Row</td>
<td>Statement Name</td>
<td>Type</td>
<td>Input</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------------------------------</td>
<td>--------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>4</td>
<td>Employee/@ID to ID</td>
<td>Map</td>
<td>@id</td>
<td>Maps the employee ID to the employee ID.</td>
</tr>
<tr>
<td>5</td>
<td>StartDate to YearofService</td>
<td>Map</td>
<td>year-from-date(curent-date) - year-from-date(start-date)</td>
<td>Determines the number of years of service by subtracting a start-date from the current-date.</td>
</tr>
</tbody>
</table>
Command Line Interface

This chapter includes the following topics:
- Command Line Interface Overview, 42
- CM_console, 42

Command Line Interface Overview

You can run a Data Transformation service from the command line of the machine that hosts the service.

Export a Data Processor transformation as a service to the `/ServiceDB` directory on the machine where you want to run the Data Transformation service. Run the CM_console command.

CM_console

Runs a Data Transformation service.

The CM_console command uses the following syntax:

```
CM_console <ServiceName>
[< -f | -u | -t >InputDocument]
[ -aServiceParameter=InitialValue]
[ -o<[Path]FileName | FileName>]
[ -r<curr | res | spec=OutputDirectory | guid>]
[ -lUserName -pPassword]
[ -v]
[ -s]
[ -x<f | u | t>InputPortName=InputDocument]
[ -x<oOutputPortName=OutputDocument]
[ -e]
```

**Note:** Do not include a space between an option and its argument.
The following table describes CM_console options and arguments:

<table>
<thead>
<tr>
<th>Option</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>ServiceName</td>
<td>Required. Specifies the name of the service.</td>
</tr>
<tr>
<td>-f</td>
<td>InputDocument</td>
<td>Optional. Specifies a path and file name on the local file system. By default, the service uses the document defined in the <code>example_source</code> property of the startup component.</td>
</tr>
<tr>
<td>-t</td>
<td>InputDocument</td>
<td>Optional. Specifies a string surrounded by double quotes.</td>
</tr>
<tr>
<td>-u</td>
<td>InputDocument</td>
<td>Optional. Specifies a URL.</td>
</tr>
<tr>
<td>-a</td>
<td>ServiceParameter=InitialValue</td>
<td>Optional. Specifies an input parameter for the service. ServiceParameter is the name of a variable as defined in the service. InitialValue must be of a data type that is valid for the defined variable. You can enter multiple input parameters, separated by spaces.</td>
</tr>
<tr>
<td>-o</td>
<td>FileName [Path]FileName</td>
<td>Optional. Directs output to Path/FileName. If you enter only FileName, you must define the Path with the -r option. By default, the CM_console command directs output to the screen.</td>
</tr>
<tr>
<td>-r</td>
<td>curr</td>
<td>Optional. Specifies the directory from which you ran the CM_console command.</td>
</tr>
<tr>
<td>-r</td>
<td>res</td>
<td>Optional. Specifies the <code>results</code> subdirectory under the directory that holds the service in the filesystem repository.</td>
</tr>
<tr>
<td>-r</td>
<td>spec=OutputDirectory</td>
<td>Optional. Specifies a directory on the local file system.</td>
</tr>
<tr>
<td>-r</td>
<td>guid</td>
<td>Optional. Specifies a directory with a unique name under the CMReports/tmp directory. You can use the configuration editor to change the location of this directory.</td>
</tr>
<tr>
<td>-l</td>
<td>UserName</td>
<td>Required when you use HTTP authentication. Specifies the user name for HTTP authentication. <strong>Note:</strong> This option is a lower-case L.</td>
</tr>
<tr>
<td>-p</td>
<td>Password</td>
<td>Required when you use HTTP authentication. Specifies the password for HTTP authentication.</td>
</tr>
<tr>
<td>-v</td>
<td>-</td>
<td>Optional. Displays verbose information about the Data Transformation version, the version of the Data Transformation syntax, the setup package identifier, the license, and other information.</td>
</tr>
<tr>
<td>-S</td>
<td>-</td>
<td>Required if the startup component of the service is a streamer. You must also use the -f option to define the input file.</td>
</tr>
<tr>
<td>-xf</td>
<td>InputPortName=InputDocument</td>
<td>Optional. InputPortName specifies the name of an <code>AdditionalInputPort</code> defined in the service. InputDocument specifies a path and file name on the local file system. You can enter multiple input ports, separated by spaces.</td>
</tr>
<tr>
<td>-xt</td>
<td>InputPortName=InputDocument</td>
<td>Optional. InputPortName specifies the name of an <code>AdditionalInputPort</code> defined in the service. InputDocument specifies a string surrounded by double quotes. You can enter multiple input ports, separated by spaces.</td>
</tr>
<tr>
<td>Option</td>
<td>Argument</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-xu</td>
<td>InputPortName=InputDocument</td>
<td>Optional. InputPortName specifies the name of an AdditionalInputPort defined in the service. InputDocument specifies a URL. You can enter multiple input ports, separated by spaces.</td>
</tr>
<tr>
<td>-xo</td>
<td>OutputPortName=OutputDocument</td>
<td>Optional. OutputPortName specifies the name of an AdditionalOutputPort defined in the service. OutputDocument specifies a path and file name on the local file system. You can enter multiple output ports, separated by spaces.</td>
</tr>
<tr>
<td>-e</td>
<td>-</td>
<td>Optional. By default, the CM_console command terminates with an exit code of 1 for success and greater than 1 for error. When you include the -e option, the CM_console command terminates with an exit code of 0 for success and greater than 1 for error.</td>
</tr>
</tbody>
</table>

For example:

```
CM_console XYZparser -fInputFile.txt -aMaxLines=1000 -oResults.xml -rcurr
```

This example calls the XYZparser service, using `InputFile.txt` as the main input document. It gives the value of 1000 to the `MaxLines` parameter, and writes the output to the `Results.xml` file in the directory from which you ran the CM_console command.
Chapter 5

Scripts

This chapter includes the following topics:

- Scripts Overview, 45
- Script Components, 45
- Script Component Properties, 48
- Script Startup Components, 49
- Example Sources, 50
- IntelliScript Editor, 51
- Sample Scripts, 52

Scripts Overview

A script performs complex transformations on input data, and writes output data. Create a script on the Objects tab of the Data Processor transformation. Use the IntelliScript editor to view a script, add and configure components, and set the startup component for a script.

Use a script to read one or more documents in any format, such as HL7, PDF, XML, or Word. You can write one or more documents in any format. You can write the output of a script to the local file system, or you can return the output through output ports of the Data Processor transformation.

A script is made up of components that define input and output documents, business logic, variables that temporarily hold data, and configuration settings. The components are arranged in a hierarchical tree. When the transformation runs a script, it begins processing in the component that you set as the startup component.

When you configure a script, you set example sources that contain sample data for each input port. When you run the transformation from the Data Viewer view, the transformation reads the example source documents. When you run the transformation in a mapping, the transformation reads the documents that it receives through its input ports.

The Data Processor transformation that contains the script must reference a schema for each XML document that the script reads or writes.

Script Components

A script component is a line or a group of lines in a script that define input and output documents, business logic, variables that temporarily hold data, and configuration settings. The components of a script appear in a hierarchical tree. Some components appear at the global level of the script, and others appear as child components.
The global level of the script contains startup components, variables, and other components such as additional input ports and transformers. A component at the global level must have a name.

A component can have properties that control the behavior of the component. The properties of a component appear nested within it. A property can appear on one line, or it can appear as a hierarchy of properties. You can configure the properties of some components to override default settings that apply to the Data Processor transformation.

Some components, such as parsers or mappers, can contain other components such as transformers or RunParser actions. Optionally, you can configure the name property of a child component.

**Component Types**

The context of the script determines the types of components you can add.

For example, anchors must appear nested within parsers, mappers, or serializers. Also, additional input ports and additional output ports can appear only at the global level of the script.

The following table describes the types of components that you can add to a script:

<table>
<thead>
<tr>
<th>Component Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Takes data from a data holder and performs an operation on it. For example, the RunParser action runs a parser.</td>
</tr>
<tr>
<td>Anchor</td>
<td>Identifies a section of the input document.</td>
</tr>
<tr>
<td>Document processor</td>
<td>Performs a complex transformation on an input document. For example, the PdfToTxt document processor converts a PDF document to plain text.</td>
</tr>
<tr>
<td>Format</td>
<td>Defines the format of the documents for a parser to process.</td>
</tr>
<tr>
<td>Locator</td>
<td>Isolates a single occurrence of a multiple-occurrence data holder.</td>
</tr>
<tr>
<td>Mapper</td>
<td>Reads XML documents and writes XML documents. Can be set as the startup component.</td>
</tr>
<tr>
<td>Notification</td>
<td>Writes a message to the standard output or to a log. For example, the XsdValidationErrors notification indicates that the input document is not valid as against the schema that defines it.</td>
</tr>
<tr>
<td>Parser</td>
<td>Reads documents in any format and writes documents in any format. Can be set as the startup component.</td>
</tr>
<tr>
<td>Script Port</td>
<td>Defines an input or output document.</td>
</tr>
<tr>
<td>Serializer</td>
<td>Reads XML documents and writes documents in any format. Can be set as the startup component.</td>
</tr>
<tr>
<td>Streamer</td>
<td>Breaks large input files into chunks and passes the chunks to a parser, mapper, or serializer. Can be set as the startup component.</td>
</tr>
<tr>
<td>Transformer</td>
<td>Transforms an input string to an output string. Can be set as the startup component.</td>
</tr>
<tr>
<td>Component Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Validator</td>
<td>Determines whether input data conforms to a specific data definition.</td>
</tr>
<tr>
<td>Variable</td>
<td>Holds data that the script receives through a service parameter, or holds data from component in the script.</td>
</tr>
</tbody>
</table>

Component Names

The name of a component identifies it in the script, the Data Processor Events view and the log.

When the Data Processor transformation performs the instructions in a component, the component generates an event that appears in the Data Processor Events view and the log. A component that appears at the global level of the script must have a name. A component that appears as the child of another component can have a name that you configure with the name property.

The name of a component must begin with a letter, must contain only English characters (A-Z, a-z), numerals (0-9), and underscores (_), and must contain no more than 127 characters.

Adding a Global Component

Define a component globally when you need to use it in two or more places in the script, or when the component can only appear at the global level.

1. At the bottom of the global level of the script, double-click the left ellipsis (...).
   A text box appears.
2. Enter the name of the component, and then press ENTER.
3. Double-click the right ellipsis.
   A list box appears.
4. Click the down arrow and select the type of component that you want to add.
   The global component appears in the script.
5. Set the properties of the component, if any.

Adding a Local Component

Define a component locally when you plan to use it in only one location in the script, or when the component can only appear as a child component.

1. At the place in the script where you want to insert a component, double-click the ellipsis.
   A list box appears.
2. Click the down arrow at the right of the list box.
   A list of available components appears, including named global components.
3. Select a component.
   The component appears in the script.
4. Set the properties of the component, if any.
Script Component Properties

The properties of a script component define the component functionality. A component can have one or more properties. The properties appear nested within the component. All components of the same type have the same properties.

For example, the \texttt{example_source} property of a parser defines sample text that the parser uses when you run the transformation from the \texttt{Data Viewer} view.

Simple Properties

The simple properties of a component are the properties that the IntelliScript editor always displays. Most users need to modify only the simple properties.

The following figure shows the simple properties of a \texttt{Parser} component:

```plaintext
myParser = Parser >>
  example_source = ...
  format = CustomFormat()
    contains ...

Advanced Properties

The advanced properties of a component are normally set to default values that users normally do not change. The IntelliScript editor normally displays only the advanced properties that you have set to a non-default value.

To show properties that are not displayed, click the double right arrow on the first line.

The following figure shows all the properties of a \texttt{Parser} component:

```plaintext
myParser = Parser <<<
  reject_recurring_pages
  no_initial_phase
  example_source = ...
  format = CustomFormat()
    sources_to_extract = ...
    serialization_mode = Full
    name = ""
    remark = ""
    on_fail = ...
    example_values = []
    source = []
    target = []
    notifications = []
      contains ...

Component Property Values

You set the values for the properties of a component.
When the value of a property is Boolean, it appears as a check box to the left of the property name. For example, the `optional` property of a `Content` component is Boolean.

When the value is a string, it appears to the right of the property name, surrounded by double quotes. Valid values are strings of valid alphanumeric characters, symbols, or control characters, but not including null characters. To enter a non-keyboard character in a text field, press `CTRL+A`, and then type the three-digit decimal code for the character. For example, type `CTRL+A 010` for a line feed, or `CTRL+A 255` for the Icelandic letter "thorn" (þ). For example, the value of the `expression` property of a `CalculateValue` component is a string.

When the value is a selection, it appears to the right of the property name. When you edit the value, a list box appears. For example, the `val_type` property of a `Variable` component is a selection.

When the value is a hierarchical tree of properties, it appears to the right and below the property name. For example, when you set the `format` property of a `Parser` component to `CustomFormat`, a tree of additional properties appears.

The following figure shows the `format` property of a `Parser` component, which appears as a tree:

```
myParser = Parser
  example_source = ...
  format = CustomFormat >>
    delimiters = DelimiterHierarchy
      delimiter_hierarchy ...
    pre_processor = ...
    default_transformers = ...
  contains ...
```

### Script Startup Components

The startup component of a script defines the entry point where the Data Processor transformation begins to process the script. The startup component must appear at the global level of the script.

You can set a parser, mapper, serializer, streamer, or transformer as the startup component.

You can set the startup component from the `Overview` tab of a Data Processor transformation. When you use the IntelliScript editor to set the startup component of a script, the script startup component becomes the Data Processor transformation startup component.

### Setting the Startup Component with the IntelliScript Editor

You can use the IntelliScript editor to set a script component as the startup component of the Data Processor transformation. You must set the startup component to run the script. You must set the startup component to display the example source in the `Input` panel of the `Data Viewer` view.

1. Open a script in the IntelliScript editor.
2. Right-click a component that appears at the global level of the script, and then select `Set as Startup Component`. 
Example Sources

An example source is a document that contains sample input data for the script to process during design time. You configure an example source for each parser, mapper, serializer, or additional input port. The example source contains the same type of data that the Data Processor transformation receives from an input port.

By default, the Data Viewer view displays the example source defined for the startup component. You can also view the example source of any other component that defines an example source. When you run a script from the Data Viewer view, the Data Processor transformation reads the example source documents.

You can configure the following types of example source document:

- LocalFile. A file on the local file system.
- Text. A string hard-coded into the script.
- URL. A file on the local network or the Internet.

Note: When you run a script in a mapping and an input document is missing, the transformation uses the example source. If no example source is configured and there is no input document, the Data Processor transformation halts and generates a fatal error.

Example Source Highlighting

The Input panel of the Data Viewer view highlights parts of the example source document.

The Data Viewer view uses different colors to highlight the content anchors of the example source, marker anchors that define where the transformation finds content, and repeating groups of anchors.

Setting an Example Source in the IntelliScript Editor

When you run a script from the Data Viewer view, you must have an example source for the main input and for each additional input port. Set the example source in the IntelliScript editor. You can also select the example source when you create a script in the Data Processor transformation.

1. Select the component for which you want to define an example source, and expand it to show its properties.
2. Next to the example_source property, double-click the ellipsis.
3. Select one of the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalFile</td>
<td>The file_name property appears under the example_source property. Double-click the ellipsis, and then browse to a file on the local file system.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Text</td>
<td>The <code>quote</code> property appears under the <code>example_source</code> property. Enter a string.</td>
</tr>
<tr>
<td>URL</td>
<td>The <code>stable_url</code> property appears under the <code>example_source</code> property. Enter a string.</td>
</tr>
</tbody>
</table>

**Viewing an Example Source**

You can view the example source of a parser, mapper, serializer, or additional input port in the **Input panel of the Data Viewer view**.

1. Open a script in the IntelliScript editor.
2. Set the one of the components of the script as the startup component.
3. In the IntelliScript editor, select the component that has the example source that you want to view.
4. In the **Data Viewer** view, click **Synchronize with Editor**.

**IntelliScript Editor**

The IntelliScript editor is a graphical tool that you use to edit scripts. Use the IntelliScript editor to add components to the script, configure component properties, and set the startup component.

When you open a script object, the IntelliScript editor appears in the editor area at the center of the Developer tool interface. By default, the IntelliScript editor displays scripts in Intelli Mode, which displays the script in an expandable hierarchical tree format, or Script Mode, which displays the script as text. You can view or edit a script in Intelli Mode. Some advanced properties are hidden by default, but you can show them by clicking a graphical double arrow on the first line of the component.

You can insert only components that are valid for the context. You can drag a component to move it, or you can cut and paste it with **CTRL+C** and **CTRL+V**. You can select multiple components with mouse clicks and the **CTRL** and **SHIFT** keys.

When you use the IntelliScript editor, the following views display relevant information:

- **Data Viewer, Input panel.** Displays the example source for the startup component or the component selected in the IntelliScript editor.
- **Data Viewer, Output panel.** Displays the output when you run the Data Processor transformation from the **Data Viewer** view.
- **Data Processor Events.** Displays the events that occur when you run a Data Processor transformation. Use the **Data Processor Events** view for troubleshooting.
- **Data Processor Script Help.** Displays documentation relevant to the component or property currently selected in the IntelliScript editor.
- **Data Processor Hex Source.** Displays the example source document in hexadecimal form. Use the **Data Processor Hex Source** view to find non-printing characters such as tabs.

To view the source of a script, right-click in the IntelliScript editor, and then select **Script Mode**. To return to Intelli Mode, right-click in the IntelliScript editor, and then select **Intelli Mode**.
## Sample Scripts

Informatica provides sample scripts as examples of tasks that you can accomplish with a script.

You can find the sample scripts in the following subdirectory of the installation directory:

\DataTransformation\samples\Projects

To view, modify, or copy a sample script, you must first import it.

The following table describes the sample scripts:

<table>
<thead>
<tr>
<th>Script Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternatives</td>
<td>Demonstrates branching and the Alternatives anchor.</td>
</tr>
<tr>
<td>AppendListItems</td>
<td>Concatenates strings in a multiple-occurrence data holder and demonstrates the AppendListItems action.</td>
</tr>
<tr>
<td>CalculateValue</td>
<td>Performs a complex numerical computation and demonstrates the CalculateValue action.</td>
</tr>
<tr>
<td>CombineValues</td>
<td>Concatenates strings and demonstrates the CombineValues and DumpValue actions.</td>
</tr>
<tr>
<td>Content</td>
<td>Demonstrates the Content anchor and the finding content in the source document by searching for a specific string, by calculating an offset from the last anchor, and by searching for an attribute in a name=value pair.</td>
</tr>
<tr>
<td>CopyValue</td>
<td>Copies an entire complex XML element with the Map action.</td>
</tr>
<tr>
<td>DelimitedSections</td>
<td>Demonstrates the DelimitedSections anchor in a parser.</td>
</tr>
<tr>
<td>DocumentOrder</td>
<td>Demonstrates branching and the Alternatives anchor, with the selector option set to DocumentOrder.</td>
</tr>
<tr>
<td>Dynamic_And_RepeatingGroup</td>
<td>Iterates over the lines of a document and demonstrates the RepeatingGroup anchor. Reads data from another location in the document based on content in the current scope.</td>
</tr>
<tr>
<td>EmbeddedParser</td>
<td>Uses an embedded secondary parser to parse the content of the main parser and demonstrates the EmbeddedParser anchor.</td>
</tr>
<tr>
<td>EnsureCondition</td>
<td>Evaluates a Boolean JavaScript expression to select alternatives and demonstrates the EnsureCondition action.</td>
</tr>
<tr>
<td>ManualSerializer</td>
<td>Demonstrates a custom serializer.</td>
</tr>
<tr>
<td>Markers</td>
<td>Demonstrates Marker anchors that use the TextSearch, OffsetSearch, TypeSearch, and PatternSearch options.</td>
</tr>
<tr>
<td>Marking_Mode</td>
<td>Demonstrates multiple methods of configuring Marker anchors.</td>
</tr>
<tr>
<td>Script Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NonMarker</td>
<td>Demonstrates a parser that uses only Content anchors and searching backward through the input document.</td>
</tr>
<tr>
<td>Pattern</td>
<td>Demonstrates extraction of data that matches a restriction defined in the schema.</td>
</tr>
<tr>
<td>persistent_search</td>
<td>Demonstrates the on_partial_match property of a Group and the adjacent property of a Marker.</td>
</tr>
<tr>
<td>ResetListVariable</td>
<td>Resets a list variable using a targetLocator.</td>
</tr>
<tr>
<td>RunSerializer</td>
<td>Demonstrates a parser that calls a secondary serializer.</td>
</tr>
<tr>
<td>HL7</td>
<td>Converts an HL7 file to XML.</td>
</tr>
<tr>
<td>TabDelimited</td>
<td>Converts a tab-delimited HL7 file to XML.</td>
</tr>
<tr>
<td>Splitter</td>
<td>Splits a file into two files, and demonstrates the WriteValue action.</td>
</tr>
<tr>
<td>TransformByParser</td>
<td>Uses a parser to transform specific text to carriage return line feed and demonstrates the TransformByParser action.</td>
</tr>
<tr>
<td>Transformers_Example</td>
<td>Demonstrates the Content anchor with the value property set to LearnByExample.</td>
</tr>
</tbody>
</table>

**Importing a Sample Script**

Import a sample script to view it or to copy parts into another script.

1. Click **File > Import**.
   - The **Import** dialog box appears.
2. Select **Informatica > Import DT Service**, and then click **Next**.
   - The **Import DT Service** page appears.
3. Next to the **Service file** field, click the **Browse** button and browse to the CMW file for the service.
4. Click **Finish**.
   - The sample script appears in a Data Processor transformation.
Parsers Overview

Parsers are script components that read source documents in any format.

The output of a parser is always XML. The input can have any format, such as text, HTML, Word, PDF, or HL7. The input can be an XML document that the parser processes as string data.

Platform-Independent Parsers

Parser scripts run on Microsoft Windows and UNIX systems. Most parser features run equally well on both platforms.

There are a few exceptions to this rule. If you plan to run a parser on Windows and UNIX, here are a few tips that can help ensure platform independence.

Newline Markers

Avoid defining Marker anchors that search for a newline character followed by a carriage return character (\n\r). This combination is commonly used in Windows but often not in UNIX.

Instead, configure a Marker with the built-in NewlineSearch component, which searches for both the \n\r sequence and the \n or \r character alone.

File Paths

Use relative, as opposed to absolute, file paths. Remember that file paths on UNIX are case-sensitive.
A **Parser** component converts a source document to XML.

**Parser**

A parser reads a source document in any format. You can add child components to perform transformations on the data.

Define parsers at the global level of the script. Set a main parser as the startup component. Call a secondary parser with the **RunParser** action. For more information, see “RunParser” on page 215.

The properties of the **Parser** appear above the *contains* line. Below the line, you can insert child components such as anchors and actions.

The following table describes the properties of the **Parser** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>example_source</td>
<td>Defines a sample source document to process in the development environment. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Empty. The Developer tool prompts you for a source document when you run the parser.</td>
</tr>
<tr>
<td></td>
<td>- InputPort. Defines an input port.</td>
</tr>
<tr>
<td></td>
<td>- LocalFile. Defines a file on the local file system.</td>
</tr>
<tr>
<td></td>
<td>- Text. Defines a string.</td>
</tr>
<tr>
<td></td>
<td>- URL. Defines a URL.</td>
</tr>
<tr>
<td></td>
<td>Default is empty.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> If the <em>sources_to_extract</em> property is set, the <em>example_values</em> property is ignored in the design environment.</td>
</tr>
<tr>
<td>example_values</td>
<td>Defines simulated values that another transformation might pass to the parser. Use this property to design a parser that is called by another parser. A parser uses the <em>example_values</em> property only when it processes the example source. It ignores the property when it parses a source document.</td>
</tr>
<tr>
<td></td>
<td>In the nested <strong>ExampleValue</strong> components, specify the data holders that the calling parser passes to this parser and their simulated values.</td>
</tr>
<tr>
<td>ExampleValue</td>
<td>Defines an example value under the <em>example_values</em> property.</td>
</tr>
<tr>
<td>format</td>
<td>Defines the format of the source document. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- BinaryFormat</td>
</tr>
<tr>
<td></td>
<td>- CustomFormat</td>
</tr>
<tr>
<td></td>
<td>- HtmlFormat</td>
</tr>
<tr>
<td></td>
<td>- Rtf Format</td>
</tr>
<tr>
<td></td>
<td>- TextFormat</td>
</tr>
<tr>
<td></td>
<td>- XmlFormat</td>
</tr>
<tr>
<td></td>
<td>Default is CustomFormat. For more information, see “Format Component Reference” on page 80.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the <em>name</em> property to identify the component that caused the event.</td>
</tr>
<tr>
<td>no_initial_phase</td>
<td>Determines whether the script searches for nested anchors in the main phase. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Search for nested anchors according to their individual properties.</td>
</tr>
<tr>
<td></td>
<td>- Selected. Search for nested anchors in the main phase.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>notifications</td>
<td>Defines a list of NotificationHandler components that the parser runs on notifications triggered by nested components. For more information, see “Notifications” on page 307.</td>
</tr>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Take no action.</td>
</tr>
<tr>
<td></td>
<td>- CustomLog. Write to the user log.</td>
</tr>
<tr>
<td></td>
<td>- LogError. Write an error message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogInfo. Write an information message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogWarning. Write a warning message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- NotifyFailure. Send a notification.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>reject_recurring_pages</td>
<td>Determines the number of times the parser parses the same page. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The parser parses a page only once.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The parser parses a page each time it follows a link to the page.</td>
</tr>
<tr>
<td></td>
<td>Use reject_recurring_pages when a web site contains many links to the same page.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> The ResetVisitedPages action resets the history list and allows a parser to process a page again, even if reject_recurring_pages is selected.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>serialization_mode</td>
<td>Defines how the script processes portions of the example source that the parser does not output to XML, when you create a serializer from a parser. For more information, see “Controlling How the Create Serializer Command Works” on page 227.</td>
</tr>
<tr>
<td></td>
<td>You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Full. Causes the Create Serializer command to copy the non-XML text to the serializer configuration.</td>
</tr>
<tr>
<td></td>
<td>- Outline. Causes the Create Serializer command to copy only the delimiters of the non-XML text to the serializer configuration. When Outline is selected, you can set the use_markers property.</td>
</tr>
<tr>
<td>source</td>
<td>Defines a sequence of data holders for input to the parser. Each data holder is identified by one of the following properties:</td>
</tr>
<tr>
<td></td>
<td>- Locator. Identifies a single-occurrence or a multiple-occurrence data holder. For multiple-occurrence data holders, each iteration accesses a new occurrence.</td>
</tr>
<tr>
<td></td>
<td>- LocatorByKey. Identifies a multiple-occurrence data holder by key.</td>
</tr>
<tr>
<td></td>
<td>- LocatorByOccurence. Identifies a multiple-occurrence data holder by sequence number.</td>
</tr>
<tr>
<td></td>
<td>In a secondary parser, set Parser &gt; source &gt; Locator &gt; data_holder to the data holder defined in the associated AdditionalInputPort &gt; data_holder. For more information, see “Source Property” on page 259.</td>
</tr>
<tr>
<td>sources_to_extract</td>
<td>Defines a hard-coded list of source documents that the parser processes. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- DocList. Defines a list of LocalFile, Text, and URL components.</td>
</tr>
<tr>
<td></td>
<td>- Empty. The parser processes the example_source.</td>
</tr>
<tr>
<td></td>
<td>- FileSearch. Defines a folder on the local file system and a file name filter.</td>
</tr>
<tr>
<td></td>
<td>- InputPort. Defines an input port. Do not use this option.</td>
</tr>
<tr>
<td></td>
<td>- LocalFile. Defines a file on the local file system.</td>
</tr>
<tr>
<td></td>
<td>- Text. Defines a string.</td>
</tr>
<tr>
<td></td>
<td>- URL. Defines a URL.</td>
</tr>
<tr>
<td></td>
<td>Default is empty.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Use the sources_to_extract property only in the design environment.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| target     | Defines a sequence of data holders for output from the parser. If the data holder does not yet exist, the parser creates it. Each data holder is identified by one of the following properties:  
- Locator. Identifies a single-occurrence or a multiple-occurrence data holder. For multiple-occurrence data holders, each iteration creates a new occurrence.  
- LocatorByKey. Identifies a multiple-occurrence data holder by key.  
- LocatorByOccurence. Identifies a multiple-occurrence data holder by sequence number.  
Use the target property when the output of the parser is used by another component. For more information, see “Target Property” on page 262. |
| use_markers| Determines whether the Create Serializer command copies the content of the Marker anchors but only the delimiters of other non-XML text. use_markers is an option under the serialization_mode property when outline is selected. Default is selected. |
This chapter includes the following topics:

- Script Ports Overview, 58
- Script Port Component Reference, 58

Script Ports Overview

A script port specifies the input or output of a script, such as a source document or an output document.

For example, in a Parser component, the values of the example_source and sources_to_extract properties are input ports.

In some components, the script ports are implicitly defined. For example, the default output file of a parser is the output.xml file. You do not need to define an output port that refers to the output.xml file.

By default, each script has one input port and one output port. You can configure additional input and output ports. When you create additional input or output ports in a script, the Developer tool adds additional ports in the Data Processor transformation.

Script Port Component Reference

A script port component specifies an input or output of a transformation, such as a source document or an output document.

AdditionalInputPort

The AdditionalInputPort port defines an additional input port.
The following table describes the properties of the AdditionalInputPort port:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code_page</td>
<td>Determines the input encoding for the port. When no value is set, the AdditionalInputPort uses the input encoding defined in the Data Processor transformation settings. Default is blank.</td>
</tr>
<tr>
<td>data_holder</td>
<td>Defines a data holder where the port stores the content of the input document. Use the same data holder in Parser &gt; source &gt; Locator &gt; data_holder property of the associated secondary parser, mapper, or serializer.</td>
</tr>
</tbody>
</table>
| disabled          | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| encode_as_xml     | Determines whether special characters are converted to XML entities. You can choose one of the following options:  
- Selected. Special characters are converted to XML entities.  
- Cleared. Special characters are not converted.  
The encode_as_xml property is a child of the input_encoding property when it is set to PortEncoding. Default is cleared. |
| example_source    | Defines the location of a source to process during testing. You can choose one of the following options:  
- LocalFile. Defines a file on the local computer.  
- Text. Defines a string.  
- URL. Defines the URL of a web page.  
Caution: Do not define a document processor under the AdditionalInputPort > example_source > pre_processor property. Define it under AdditionalInputPort > pre_processor. |
| input_encoding    | Defines the encoding of the input.                                                                                                                                 |
| PortEncoding      | Defines custom settings for the additional input. The PortEncoding property has the following options:  
- code_page  
- encode_as_xml |
| pre_processor     | Defines the name of a document processor to apply to the input before the document processor defined under RunParser > pre_processor. For more information, see “Document Processor Component Reference” on page 67.  
Caution: Do not define a document processor under the AdditionalInputPort > example_source > pre_processor property. Define it under AdditionalInputPort > pre_processor. |

Define the AdditionalInputPort port at the global level of the script and assign it a name.

### Example of AdditionalInputPort

Suppose you have two text files:

- IdsAndSalaries.txt is a table of employee IDs and salaries.
- IdsAndNames.txt is a table of employee IDs and names.

You want to parse these files jointly, generating an XML output file containing the employee names and salaries. You can configure the transformation in the following way:

- The main parser, called EmployeeParser, processes IdsAndSalaries.txt.
• The main parser activates a secondary parser, called `IdsToNamesParser`, which processes `IdsAndNames.txt` and stores the result in an XML table.

• The main parser uses a `LookupTransformer` to convert the IDs to names. The lookup table is the output of the secondary parser.

The following script illustrates this configuration. The secondary parser references an `AdditionalInputPort` that retrieves the `IdsAndNames.txt` file.

```plaintext
[Script]
```

---

**AdditionalOutputPort**

The `AdditionalOutputPort` port defines an additional output port. Use this component to define output in multiple locations or multiple documents.

The following table describes the properties of the `AdditionalOutputPort` port:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_BOM_prefix</td>
<td>Adds a byte-order mark (BOM) prefix to the output. The type of BOM prefix is determined by the output encoding defined in the <code>output_encoding</code> property. Default is cleared.</td>
</tr>
<tr>
<td>code_page</td>
<td>Defines the <code>encoding</code> attribute of the additional output. If this property is not set, the additional output is generated with the output encoding defined in the Data Processor transformation settings. The <code>code_page</code> property is a child of the <code>output_encoding</code> property when it is set to <code>PortEncoding</code>. Default is cleared.</td>
</tr>
</tbody>
</table>
| disabled       | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
### Property | Description
--- | ---
`encode_as_xml` | Determines whether special characters are converted to XML entities. You can choose one of the following options:
- Selected. Special characters are converted to XML entities.
- Cleared. Special characters are not converted. The `encode_as_xml` property is a child of the `output_encoding` property when it is set to `PortEncoding`. Default is cleared.
`file_extension` | Defines the file extension for the additional output file in the design environment. The name of the file is the name assigned to the `AdditionalOutputPort` component. This setting has no effect in the production environment. Default is `.xml`.
`other_properties` | Defines encoding properties when it is set to `XmlHeader`. You can choose one of the following options:
- `XmlHeader`. Defines the XML header. The `XmlHeader` property has the following options:
  - `add_BOM_prefix`
  - `process_instruction`
  - `process_instruction_string`
  - `root_element`
  - `xml_version`
  - `XSLT_stylesheet_name`
- Cleared. Output properties are determined by the Data Processor transformation settings. Default is cleared.
`output_encoding` | Defines encoding properties when it is set to `PortEncoding`. You can choose one of the following options:
- `PortEncoding`. The additional output has custom settings for `code_page` and `encode_as_xml`.
- Cleared. The Data Processor transformation settings control the output encoding and conversion of XML entities. Default is cleared.
`PortEncoding` | Defines custom settings for the additional output. You can choose one of the following options:
- `code_page`
- `encode_as_xml`
`process_instruction` | Defines a processing instruction in the output XML file. You can choose one of the following options:
- None. Does not write a processing instruction to the XML output.
- `UseOutputCodePage`. Outputs the code page defined in the `output_encoding` property.
- `FreeEncodingString`. Outputs the string defined in the `process_instruction_string` property. Default is `UseOutputCodePage`.
`process_instruction_string` | Defines a user-defined processing instruction. The `process_instruction_string` property has an effect only when the `process_instruction` property is set to `FreeEncodingString`.
`root_element` | Defines the name of the root element that is wrapped around the entire output.
`xml_version` | Defines the version attribute of the processing instruction. Default is 1.0.
`XSLT_stylesheet_name` | Defines an XSLT stylesheet that is written to the processing instruction.

### Defining an Additional Output Port

1. At the global level of the script, insert an `AdditionalOutputPort` component, and assign it a name.
2. Nested under the startup component of the Data Processor transformation, insert a WriteValue action, set the output property to OutputPort, and set port to the name of the additional output port.

3. In the Data Processor transformation settings, select Output Control and then check Disable Automatic Output.

**File Name of Additional Output**

When you run the transformation in the Developer tool, the system defines a file name for the additional output, and it stores the file in the results folder of the project. For example, if the port is called MyOutputPort, the file name might be `output_MyOutputPort.xml`.

To determine the file name:

1. Click Run > Run.
2. Click Details to display the I/O Ports table.
   - The table displays the name of each AdditionalOutputPort and its output file.

When you deploy the transformation as a service, an application that runs the service can pass the additional output location as a parameter. For example, the location might be a buffer.

**Example of AdditionalOutputPort**

A parser generates the following XML structure:

```xml
<Person gender="M">
  <Name>
    <First>Ron</First>
    <Last>Lehrer</Last>
  </Name>
  <Id>547329876</Id>
  <Age>27</Age>
</Person>
```

The parser uses two WriteValue actions to generate output.

The first WriteValue writes the entire `<Person>` element to the default results file.

```xml
<Person gender="M">
  <Name>
    <First>Ron</First>
    <Last>Lehrer</Last>
  </Name>
  <Id>547329876</Id>
  <Age>27</Age>
</Person>
```

The second WriteValue references an AdditionalOutputPort to write the nested `<Name>` element to another file.

```xml
<Name>
  <First>Ron</First>
</Name>
```
DocList

The DocList port defines a list of the following types of input ports:

- LocalFile
- Text
- URL

The following table describes the properties of the DocList port:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled  | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:
- Selected. The script ignores the component.
- Cleared. The script applies the component.
The default is cleared. |
| pre_processor | Defines the name of the preprocessor to apply to the input files. For more information, see “Document Processor Component Reference” on page 67. |

FileStream

The FileStream port defines input files on a computer in the local network. Use the FileStream port in the sources_to_extract property of a Parser.

The following table describes the properties of the FileStream port:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>directory</td>
<td>Defines a folder that contains the input files.</td>
</tr>
</tbody>
</table>
| disabled  | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:
- Selected. The script ignores the component.
- Cleared. The script applies the component.
The default is cleared. |
| pre_processor | Defines the name of a document processor to apply to the input files. For more information, see “Document Processor Component Reference” on page 67. |
| recursive | Determines whether the input files can occur in subfolders of the specified folder. Default is cleared. |
| wildcard | Defines a criterion for filtering the files in the specified folder. Use * as a wildcard character. For example, *.txt finds all TXT files. Default is *.txt. |

InputPort

The InputPort port defines a named input port that is defined with the AdditionalInputPort component.
The following table describes the properties of the **InputPort** port:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>Defines the name of the AdditionalInputPort component that defines the input.</td>
</tr>
</tbody>
</table>

**LocalFile**

The **LocalFile** port defines a file on the local network.

The following table describes the properties of the **LocalFile** port:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_name</td>
<td>Defines the path and filename of a file on the local network.</td>
</tr>
<tr>
<td>pre_processor</td>
<td>Defines the name of a document processor to apply to the file. For more information, see “Document Processor Component Reference” on page 67.</td>
</tr>
<tr>
<td>simulated_url</td>
<td>Defines a URL to assign to the file. This property causes the parser to treat the file as if it were located on a web server. If the file contains relative links, the parser resolves the links relative to the URL. The host name portion of the URL is not case sensitive.</td>
</tr>
</tbody>
</table>

**OutputPort**

The **OutputPort** port defines a named output port that is defined with the AdditionalOutputPort component. You can use an **OutputPort** port in a **WriteValue** action.

The following table describes the properties of the **OutputPort** port:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>port</td>
<td>Determines the name of the AdditionalOutputPort.</td>
</tr>
</tbody>
</table>

**Text**

The **Text** port defines a text string that is used as input of a transformation.

The following table describes the properties of the **Text** port:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre_processor</td>
<td>Defines the name of a document processor to apply to the string. For more information, see “Document Processor Component Reference” on page 67.</td>
</tr>
<tr>
<td>quote</td>
<td>Defines a text string.</td>
</tr>
</tbody>
</table>
URL

The **URL** port defines the URL of a document that is available on a web server.

The following table describes the properties of the URL port:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>simulated_url</td>
<td>Defines a URL to assign to the string. This property causes the parser to treat the string as if it were a file located on a web server. If the string contains relative links, the parser resolves the links relative to the URL.</td>
</tr>
<tr>
<td>size</td>
<td>Defines a static size for the text buffer. Use the <code>size</code> property with binary sources. Default is -1, which means that the buffer is dynamically sized.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>post_data</td>
<td>Defines data that the transformation posts to the URL.</td>
</tr>
<tr>
<td>pre_processor</td>
<td>Defines the name of a document processor to apply to the files.</td>
</tr>
<tr>
<td>retries</td>
<td>Defines the number of retries that the parser performs before it reports a failure. Default is 0.</td>
</tr>
<tr>
<td>seconds_to_wait</td>
<td>Defines the number of seconds to wait between retries. Default is 60.</td>
</tr>
<tr>
<td>stable_url</td>
<td>Defines a URL address that contains an input document.</td>
</tr>
</tbody>
</table>
Document Processors

This chapter includes the following topics:

- Document Processors Overview, 66
- Defining a Document Processor, 66
- Document Processor Component Reference, 67
- TextML XML Schema, 74
- PdfToTxt_4 Table Configuration Editor, 75

Document Processors Overview

Document processors are components that convert the format of a complete document to another format for processing.

You can use a document processor as a pre-processor that converts the format of a source document before a transformation. For example, if the source document of a parser is in the PDF format, you might apply the PdfToTxt_4 processor. This converts the source document to text, which is much easier to parse than the binary PDF format.

Do not confuse document processors with format preprocessors. For more information about format preprocessors, see “Formats Overview” on page 79.

Defining a Document Processor

You can pre-process the source document with any document processor.

1. Assign the example_source property of the transformation. The value of the example_source is an input port, such as LocalFile or Text.

2. Assign the pre_processor property of the input port.

   The script applies the processor that you define under example_source to all sources on which you run the transformation.

   **Note:** You can also define a pre-processor in the sources_to_extract property of a parser. The processor that you define there applies only to the source documents that you define in sources_to_extract, and not to any other document that the parser processes.
Display of Document Processor Output

If you assign a document processor to the example source, the source panel of the Data Viewer view displays the processor output.

Document Processor Component Reference

Document processors convert a complete document from one format to another before it is processed by a parser, mapper, or serializer.

AFPToXML

The AFPToXML document processor converts the IBM Advanced Function Presentation print-stream format to XML.

The processor output is in the UTF-8 encoding. If a transformation receives input from the processor, you must set the input encoding to UTF-8.

AsnToXml

The AsnToXml document processor converts a binary ASN.1 file to XML.

The following table describes the properties of the AsnToXml document processor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asn_file</td>
<td>Defines an ASN.1 specification file.</td>
</tr>
<tr>
<td>header</td>
<td>Defines a header to exclude from the XML. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- NewlineSearch. The header is a newline.</td>
</tr>
<tr>
<td></td>
<td>- OffsetSearch. The header is defined by the number of characters from the beginning of the file.</td>
</tr>
<tr>
<td></td>
<td>- PatternSearch. The header is defined by a regular expression.</td>
</tr>
<tr>
<td></td>
<td>- TextSearch. The header is defined by an explicit string or a string that you retrieve dynamically from the source document.</td>
</tr>
<tr>
<td>no_constraints</td>
<td>Determines whether the ASN file is processed with constraints. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- true. The ASN file is processed without constraints.</td>
</tr>
<tr>
<td></td>
<td>- false. The ASN file is processed with constraints.</td>
</tr>
<tr>
<td></td>
<td>Default is false.</td>
</tr>
<tr>
<td>pdu_type</td>
<td>Defines the PDU type. Use this property to clarify an ambiguity.</td>
</tr>
</tbody>
</table>
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>process_first_message</td>
<td>Determines whether the entire CDR file is processed. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- true. Only the first record is processed.</td>
</tr>
<tr>
<td></td>
<td>- false. The entire CDR file is processed. Default is false.</td>
</tr>
<tr>
<td>separator</td>
<td>Defines text to ignore between records. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- NewlineSearch. The separator is a newline.</td>
</tr>
<tr>
<td></td>
<td>- OffsetSearch. The separator is defined by the number of characters from the end of the previous record.</td>
</tr>
<tr>
<td></td>
<td>- PatternSearch. The separator is defined by a regular expression.</td>
</tr>
<tr>
<td></td>
<td>- TextSearch. The separator is defined by an explicit string or a string that you retrieve dynamically from the source document.</td>
</tr>
</tbody>
</table>

**ExcelToDataXml**

The ExcelToDataXml document processor converts Microsoft Excel documents to XML.

The following table describes the properties of the ExcelToDataXml document processor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>Determines the content of the output. The enabled property has the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The output contains raw data and formatted data.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The output contains only formatted data.</td>
</tr>
<tr>
<td></td>
<td>Default is selected.</td>
</tr>
</tbody>
</table>

| param1   | Determines whether raw data appears in the output of the document processor when the raw data differs from the formatted data. param1 is named Display_raw_data_when_different and has only one property, enabled. |

The XML contains the data and the results of formulas that existed in the original Excel document. It does not preserve the formulas themselves, formatting information, or macro code. If you need to use macro code, use ExcelToXml rather than ExcelToDataXml.

The XML representation conforms to a subset of the ExcelToXml.xsd schema, which you can find in the doc subdirectory of the installation directory.

The processor output is in the UTF-8 encoding. If a transformation receives input from the processor, you must set the input encoding to UTF-8.

The processor supports Excel version 97 and later. It accesses its input directly, not through the Excel application. You do not need to install Excel on the computer. The processor supports both the XLS format and the XLSX format.

This component is implemented in Java and requires correct configuration of the Java Runtime Environment (JRE).

**ExcelToXml**

The ExcelToXml document processor converts Microsoft Excel documents to XML.
The following table describes the properties of the ExcelToXml document processor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>Defines the value of <code>param2</code> or <code>param3</code>.</td>
</tr>
<tr>
<td>param1</td>
<td>Defines the sheets of the Excel workbook to include in the XML. In the XML output, each sheet is represented by a <code>&lt;sheet&gt;</code> element. <code>param1</code> is named <code>include_sheets</code> and has only one property, <code>value</code>.</td>
</tr>
</tbody>
</table>
| param2   | Determines whether the document processor includes empty cells in the output XML. `param2` is named `include_empty_cells` and has only one property, `enabled`, which has the following options:
- Selected. The output includes empty cells.
- Cleared. The output omits empty cells.
Default is selected. |
| param3   | Determines whether the document processor includes Excel macro code in the output XML. `param3` is named `include_macro_information` and has only one property, `enabled`, which has the following options:
- Selected. The document processor includes macro code.
- Cleared. The document processor omits macro code.
Default is cleared. |
| value    | Defines a list of the following options:
- The string "All". The output includes all sheets.
- Data holders containing the sheet names. The output includes only the named sheets. If you list a sheet that does not exist in the workbook, the processor generates a `<sheet>` element containing a warning message. The other sheets are processed normally. Default is All. |

The XML preserves the data, formulas, formatting, and macro code that existed in the original Excel document. If only the data is required, use the ExcelToDataXml processor, which offers smaller output and better performance.

The XML representation conforms to the ExcelToXml.xsd schema, which is in the doc subdirectory of the installation directory.

The processor output is in the UTF-8 encoding. If a transformation receives input from the processor, you must set the input encoding to UTF-8.

The processor supports Excel version 97 and later. It accesses its input directly, not through Excel. You do not need to install Excel on the computer.

This component is implemented in Java and requires correct configuration of the Java Runtime Environment (JRE).

**ExcelToXml_03_07_10**

The ExcelToXml_03_07_10 document processor converts the following files to XML:

- XLSX files created with Microsoft Excel 2007 or 2010
- XLS files created with Microsoft Excel 2003, 2007, or 2010

**ExpandFrameSet**

The ExpandFrameSet document processor opens all the frames of an HTML document. Use this document processor when the source document of a parser is an HTML frameset. The parser runs on the content of all the frames.
**ExternalJavaPreProcessor**

The **ExternalJavaPreProcessor** document processor runs a user-defined document processor that is implemented in Java.

The following table describes the properties of the **ExternalJavaPreProcessor** document processor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jclass</td>
<td>Defines the path of the Java class.</td>
</tr>
<tr>
<td>jmethod</td>
<td>Defines the method to run.</td>
</tr>
</tbody>
</table>

This component is implemented in Java and requires correct configuration of the Java Runtime Environment (JRE).

**Note:** This component is deprecated. The IntelliScript editor displays it for legacy scripts. Do not use it in new scripts. Instead, create a custom Java document processor. For more information, see “Developing a Custom Component in Java” on page 312.

**ExternalPreProcessor**

The **ExternalPreProcessor** document processor runs a user-defined document processor that is implemented as a C++ DLL.

The following table describes the properties of the **ExternalPreProcessor** document processor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>Defines a DLL in the ExternLib\Users folder.</td>
</tr>
</tbody>
</table>

**Note:** This component is deprecated. The IntelliScript editor displays it for legacy scripts. Do not use it in new scripts. Instead, create a custom C or C++ document processor. For more information, see “Developing a Custom Component in C or C++” on page 314.

**HIPAAValidator**

The **HIPAAValidator** document processor validates HIPAA messages and generates HIPAA acknowledgments. The **HIPAA_Validation** project of the HIPAA library uses this processor.

The following table describes the properties of the **HIPAAValidator** document processor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| param1   | The param1 property is named **validation_params** and has only one property, **value**, which has the following options:  
- LDNSB  
- Validator |
<p>| param2   | Defines the validation type. The param2 property is named <strong>types_to_validate</strong> and has only one property, <strong>value</strong>. Valid values are 1 to 7. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| param3   | Defines the format for error report output. The param3 property is named report_formats and has only one property, value, which has the following options:  
- HTML. Use for display in the Developer tool.  
- XML. Use for further processing. |
| param4   | Defines the acknowledgment type. The param4 property is named generate_acknowledgments and has only one property, value, which has the following options:  
- 277  
- 824  
- 997  
- 999  
- TA1 |
| value    | Defines the value of param1, param2, param3, or param4. |

Note: This document processor operates on Windows and Linux x64 platforms. Before you can use it, you must install and configure the HIPAA validation add-on package on every computer where you run HIPAAValidator.

PdfFormToXml_1_00

The PdfFormToXml_1_00 document processor converts PDF forms to XML. The processor supports forms that conform to the Adobe AcroForms standard.

PdfToTxt_3_02

The PdfToTxt_3_02 document processor converts PDF files to text.

The following table describes the properties of the PdfToTxt_3_02 document processor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>Defines the value of param2 or param4.</td>
</tr>
<tr>
<td>param1</td>
<td>Defines a string or variable that contains the word spacing factor. The param1 property is named WordSpacingFactor and has only one property, value, which contains the string or variable. Default is 1.8.</td>
</tr>
</tbody>
</table>
| param2   | Determines whether the output document is optimized for tables. The param2 property is named OptimizeForTables and has only one property, enabled, which has the following options:  
- Selected. The output document is optimized for tables.  
- Cleared. The output document is not optimized for tables.  
Default is cleared. |
| param3   | Defines a string or variable that contains the password. The param3 property is named Password and has only one property, value, which contains the string or variable. |
| param4   | The param4 property is named HideNewPageChar and has only one property, enabled, which has the following options:  
- Selected. New page characters are hidden.  
- Cleared. New page characters are not hidden.  
Default is cleared. |
**Property** | **Description**
---|---
param5 | Defines a string or variable that contains advanced optimizations. The `param5` property is named `AdvancedOptimizations` and has only one property, `value`, which contains the string or variable.
value | Defines the value of `param1`, `param3`, or `param5`.

The `PdfToTxt` pre-processor might not support certain PDFs with embedded fonts. If the pre-processor fails, copy the text from the input PDF into Notepad to check for embedded fonts. If you cannot paste the text or if it is corrupted, the PDF probably contains embedded fonts.

**Note:** This component is deprecated. The IntelliScript editor displays it for legacy projects. Do not use it in new scripts.

### PdfToTxt_4

The `PdfToTxt_4` document processor converts PDF files to text or XML.

The following table describes the properties of the `PdfToTxt_4` document processor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| param1 | Defines the PDF table layout. The `param1` property has only one option: `PdfLayout`.
| value | Defines the PDF table layout. Double-click the `value` property to open the table configuration editor. |

The table configuration editor customizes the way tables are read. Use it to correct problems with column alignment, word wrapping, line spacing, and overflow from one cell to another. For more information, see “PdfToTxt_4 Table Configuration Editor” on page 75.

The `PdfToTxt_4` document processor generates text output by default. Use the table configuration editor to select XML output. The XML conforms to the `PDF4.xsd` schema, which you can find in the following directory:

`<INSTALL_DIR>\DataTransformation\doc`

When you use the `PdfToTxt_4` document processor, set the input encoding to UTF-8 to enable the parser, mapper, or serializer to correctly read the document.

**Note:** The `PdfToTxt` pre-processor might not support certain PDFs with embedded fonts. If the pre-processor fails, copy the text from the input PDF into Notepad to check for embedded fonts. If you cannot paste the text or if it is corrupted, the PDF probably contains embedded fonts.

### PowerpointToTextML

The `PowerpointToTextML` document processor converts Microsoft PowerPoint (PPT) presentations to the TextML XML schema. For more information, see “TextML XML Schema” on page 74.

This component supports PowerPoint version 97 and higher. It accesses its input directly, not through PowerPoint. You do not need to install PowerPoint on the computer.

This component is implemented in Java and requires correct configuration of the Java Runtime Environment (JRE).
ProcessByTransformers

The ProcessByTransformers document processor runs a transformer or a sequence of transformers on the entire document. A transformation can then run on the output of the transformers.

Define the list of transformers under the transformers line.

ProcessorPipeline

The ProcessorPipeline document processor defines a sequence of document processors to run on a document. Use this component when you need to run two or more document processors.

Define the list of document processors under the pre_processor_list line.

RtfToTextML

The RtfToTextML document processor converts RTF files to the TextML XML schema. For more information, see “TextML XML Schema” on page 74.

The processor output is in the UTF-8 encoding. If a transformation receives input from the processor, you must set the input encoding to UTF-8.

WordToXml

The WordToXml document processor converts Microsoft Word documents to XML.

The processor output is in the UTF-8 encoding. If a transformation receives input from the processor, you must set the input encoding to UTF-8.

This component supports Word version 97 and higher. It accesses its input directly, not through Microsoft Word. You do not need to install Word on the computer.

This component is implemented in Java and requires correct configuration of the Java Runtime Environment (JRE).

XmlToDocument_372

The XmlToDocument_372 document processor converts XML data to document formats, such as PDF or Excel. You can use it as a post-processor to convert parser or mapper output to various document types.

This component uses the Business Intelligence and Reporting Tool (BIRT) Eclipse add-on to generate the output documents. In BIRT, you must configure a report that converts the XML to the desired document format. The XmlToDocument_372 processor runs the report.

For more information about BIRT, see http://www.eclipse.org/birt.
The following table describes the properties of the `XmlToDocument_372` document processor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>param1</code></td>
<td>The path and file name of the BIRT <code>*.rptdesign</code> file. The <code>param1</code> property is named <code>report_file</code> and contains the property <code>value</code>, which contains the path and file name.</td>
</tr>
<tr>
<td><code>param3</code></td>
<td>A variable that holds the location of the <code>*.rptdesign</code> file. The <code>param3</code> property is named <code>report_location</code> and contains the property <code>value</code>, which points to the variable. Default is <code>$VarServiceInfo/*/s/ServiceLocation</code>.</td>
</tr>
<tr>
<td><code>value</code></td>
<td>Contains the value of <code>param1</code>, <code>param2</code>, or <code>param3</code>.</td>
</tr>
</tbody>
</table>

**Note:** Effective in version 9.5.1, the `XmlToDocument` processor is deprecated. The IntelliScript editor still displays the `XmlToDocument` preprocessor in existing scripts, but you can no longer add the preprocessor to new scripts. Use the `XmlToDocument_372` preprocessor instead.

**XmlToExcel**

The `XmlToExcel` document processor converts XML documents to Microsoft Excel format.

The processor operates on an XML representation of an Excel workbook. The XML representation must be in the UTF-8 encoding and it must conform to the `ExcelToXml.xsd` schema. You can find the schema in the `doc` subdirectory of the installation directory. The schema file is provided for your information. You can use the processor without adding the schema to your project.

The processor reverses the operation of `ExcelToXml`. For example, you can use `ExcelToXml` to convert an Excel workbook to XML. You can then alter some of the XML data and use `XmlToExcel` to convert the data back to an Excel workbook.

This component supports Excel version 97 and higher. It writes its output directly, not through Microsoft Excel. You do not need to install Excel on the computer.

This component is implemented in Java and requires correct configuration of the Java Runtime Environment (JRE).

**TextML XML Schema**

Some of the document processors convert documents to an XML vocabulary called TextML. This is a simple XML vocabulary for saving document content without layout.

The TextML schema, `textML.xsd`, is available in the `\doc` subfolder of the installation folder.
The following is a sample TextML document.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <docinfo>
    <title>TextML Sample</title>
    <author>Tex Tomiller</author>
    <company>Acme Gizmos, Inc.</company>
    <modified>2004-03-14T14:39:00</modified>
    <created>2004-03-12T09:15:00</created>
    <last_author>Tex Tomiller</last_author>
    <word_count>16</word_count>
    <char_count>105</char_count>
  </docinfo>
  <docbody>
    <p>This is a sample of the TextML XML vocabulary.</p>
    <p>TextML saves document content without layout information.</p>
  </docbody>
</document>
```

### PdfToTxt_4 Table Configuration Editor

The table configuration editor customizes the way the PdfToTxt_4 document processor converts tables in PDF documents.

Use the table configuration editor when default settings of the PdfToTxt_4 document processor do not correctly render column alignment, word wrapping, line spacing, or overflow from one cell to another.

**Note:** The user interface for the table configuration editor appears only in English.

1. Add a parser, mapper, serializer, or AdditionalInputPort to the script.
2. Under the `example_source` property, set the `pre_processor` property to PdfToTxt_4.
3. Under the `pre_processor` property, double-click the `value` property.
   - The table configuration editor appears. The upper panel displays the input PDF document, and the lower panel displays the PdfToTxt_4 output.
   - Table editing commands appear in the toolbar at the top of the window. You can right-click to display an editing menu.
4. Browse to a table in the PDF document and click **Add Table**.
   - The name of the table appears in the **Tables** field and in the **Name** field.
5. Select **Use Regular Expressions**. In the **Table Start** field, enter a regular expression that defines the upper left corner of the table.
   - **Tip:** Use the headings of the first two columns as the regular expression. Add more column headings as needed to make **Table Start** unique. Separate the headings by a single space character, even if the columns are widely separated.
6. In the **Table End** field, enter a regular expression that defines the text immediately after the table.
   - **Note:** The value of **Table End** must appear in the body of the document, not in a page footer.
7. Click **Process**.
   - The editor displays the table configuration that PdfToTxt_4 detects. The top and bottom of the table appear as horizontal blue lines. The default column borders appear as vertical red lines.
8. To edit the column borders, perform one or more of the following steps:
   ✦ Drag a column border to the right or left to change its position.
   ✦ Click Add Column to add a column.
   ✦ Click Remove Column and select a column border to delete a column.

   **Note:** If the table contains horizontally merged cells, PdfToTxt_4 might truncate the entries.

9. Examine the output window to confirm that the table is converted properly. If not, correct the table definitions.

10. Repeat steps 1-9 for each table in the PDF document.

11. Click OK to return to the Developer tool.

   An XML string that defines the table configuration appears in the value property of the PdfToTxt_4 document processor.

**Editor Options**

The following table describes the controls and fields in the PdfToTxt_4 table configuration editor.

<table>
<thead>
<tr>
<th>Control or Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoom In</td>
<td>Make the PDF display larger.</td>
</tr>
<tr>
<td>Zoom Out</td>
<td>Make the PDF display smaller.</td>
</tr>
<tr>
<td>Fit Width</td>
<td>Display the PDF document according to the width of the window.</td>
</tr>
<tr>
<td>Prev Page</td>
<td>Go to the previous page.</td>
</tr>
<tr>
<td>Next Page</td>
<td>Go to the next page.</td>
</tr>
<tr>
<td>Find</td>
<td>Search for a string in the PDF.</td>
</tr>
<tr>
<td>Add Table</td>
<td>Add a table to the configuration.</td>
</tr>
<tr>
<td>Rem. Table</td>
<td>Remove a table from the configuration.</td>
</tr>
<tr>
<td>Add Column</td>
<td>Add a column border to the current table.</td>
</tr>
<tr>
<td>Rem. Column</td>
<td>Delete the currently selected column border.</td>
</tr>
<tr>
<td>Process</td>
<td>Apply the current table definitions. Click Process after every table and column-related action to apply that action.</td>
</tr>
<tr>
<td>Tables</td>
<td>A list of tables defined in the input PDF. You can select a table by clicking it.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of the currently selected table.</td>
</tr>
<tr>
<td>Table Start</td>
<td>An expression defining the upper left corner of the table.</td>
</tr>
<tr>
<td>Table End</td>
<td>An expression defining the first text after the table.</td>
</tr>
<tr>
<td>Page Header</td>
<td>An expression defining the end of the page header. Use this option to exclude the header from the table processing.</td>
</tr>
</tbody>
</table>
PDF Conversion Example

This example illustrates the PdfToTxt_4 table configuration procedure using a sample parser project and a sample PDF document.

The processor input is a small financial report in PDF format. The report contains some text and two tables. Use the table configuration editor to ensure that the processor converts the tables correctly to text.

Configuring the First Table

1. Configure a parser and assign the PDF document as the example_source. Double-click on the value property to open the table configuration editor.
2. In the PDF display, browse to the first table.
3. Set Table Start = GID RMS ID, the headings of the first two columns of the table. Note that the expression is case sensitive.
4. Set Table End = Forward exchange transactions, the first text following the table. The editor displays the table configuration.
5. If necessary, adjust the table definition and the columns. You can drag, add, or remove column borders.
Configuring the Second Table

The second table extends over multiple pages.

1. Click Add Table.
   
   The system displays Table 2 in the Tables and Name fields.

2. Set Table Start = Ticker Shares Traded.

3. Set Table End = Conclusion, the first body text after the table.

4. Click Process to configure the table.

5. Adjust the right borders of the Shares Traded and Currency columns.

6. Perform the following steps to eliminate the page header and footer from the output document:
   
   a. Set Page Header = Gain/Loss.
   
   b. Set Page Footer = Page [1-9].
   
   c. Click Process.
This chapter includes the following topics:

- Formats Overview, 79
- Standard Format Properties, 80
- Format Component Reference, 80
- Delimiters Component Reference, 86
- Format Preprocessor Component Reference, 91

Formats Overview

The `format` property of a parser defines the format of the documents for the transformation to process. The value of the property is one of the following format components:

- BinaryFormat
- CustomFormat
- HtmlFormat
- RtfFormat
- TextFormat
- XmlFormat

The format has properties of its own, which further define how the parser interprets and processes the input. Within a format, you can nest the following subcomponents:

<table>
<thead>
<tr>
<th>Subcomponent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delimiter</td>
<td>Defines a hierarchy of characters or strings that organize the information in the document, such as newlines and tabs.</td>
</tr>
<tr>
<td>Format preprocessor</td>
<td>Cleans up the source before the parser starts searching for anchors.</td>
</tr>
<tr>
<td>Default transformer</td>
<td>Performs predefined operations on the output of each anchor.</td>
</tr>
</tbody>
</table>
# Standard Format Properties

The following table describes standard properties of the format components:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_transformers</td>
<td>Defines a list of transformers that the parser applies to the output of each content anchor.</td>
</tr>
</tbody>
</table>
| delimiters        | Defines the structure of information in the document. You can choose one of the following options:  
  - CommaDelimited. Data fields are separated by commas.  
  - DelimiterHierarchy. Data fields are separated or surrounded by text characters.  
  - HL7. Data fields are separated as defined in the HL7 standard.  
  - Positional. Data fields are defined by the number of characters between them.  
  - PostScript. Data fields are defined according to the PostScript format.  
  - RTF. Data fields are defined according to the RTF format.  
  - SGML. Data fields are defined according to the SGML format.  
  - SpaceDelimited. Data fields are separated by spaces.  
  - TabDelimited. Data fields are separated by tabs.  
  For more information, see “Delimiters Component Reference” on page 86. |
| name              | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| pre_processor     | Defines a format preprocessor that processes the input after any document processor that you defined for the pre_processor property of the example_source. You can choose one of the following options:  
  - HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character. It is not restricted to HTML documents.  
  - RtfProcessor. Normalizes RTF files.  
  Default is blank. |
| remark            | A user-defined comment that describes the purpose or action of the component. |

# Format Component Reference

Format components define the format of input documents. Define format components under the format property of a Parser.

**BinaryFormat**

The BinaryFormat format processes binary files and text files that you want to treat as a buffer of binary bytes.
The following table describes the properties of the **BinaryFormat** format:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_transformers</td>
<td>Defines a list of transformers that the parser applies to the output of each content anchor. Default is empty.</td>
</tr>
<tr>
<td>delimiters</td>
<td>Defines the structure of information in the document. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- CommaDelimited. Data fields are separated by commas.</td>
</tr>
<tr>
<td></td>
<td>- DelimiterHierarchy. Data fields are separated or surrounded by text characters.</td>
</tr>
<tr>
<td></td>
<td>- HL7. Data fields are separated as defined in the HL7 standard.</td>
</tr>
<tr>
<td></td>
<td>- Positional. Data fields are defined by the number of characters between them.</td>
</tr>
<tr>
<td></td>
<td>- PostScript. Data fields are defined according to the PostScript format.</td>
</tr>
<tr>
<td></td>
<td>- RTF. Data fields are defined according to the RTF format.</td>
</tr>
<tr>
<td></td>
<td>- SGML. Data fields are defined according to the SGML format.</td>
</tr>
<tr>
<td></td>
<td>- SpaceDelimited. Data fields are separated by spaces.</td>
</tr>
<tr>
<td></td>
<td>- TabDelimited. Data fields are separated by tabs.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “Delimiters Component Reference” on page 86.</td>
</tr>
<tr>
<td></td>
<td>Default is Positional.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>pre_processor</td>
<td>Defines a format preprocessor that processes the input after any document processor that you defined for the pre_processor property of the example_source. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character. It is not restricted to HTML documents.</td>
</tr>
<tr>
<td></td>
<td>- RtFProcessor. Normalizes RTF files.</td>
</tr>
<tr>
<td></td>
<td>Default is empty.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

**CustomFormat**

The **CustomFormat** format is a user-defined format for processing any type of source document.
The following table describes the properties of the **CustomFormat** format:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_transformers</td>
<td>Defines a list of transformers that the parser applies to the output of each content anchor. Default is empty.</td>
</tr>
</tbody>
</table>
| delimiters    | Defines the structure of information in the document. You can choose one of the following options:  
|               | - CommaDelimited. Data fields are separated by commas.  
|               | - DelimiterHierarchy. Data fields are separated or surrounded by text characters.  
|               | - HL7. Data fields are separated as defined in the HL7 standard.  
|               | - Positional. Data fields are defined by the number of characters between them.  
|               | - PostScript. Data fields are defined according to the PostScript format.  
|               | - RTF. Data fields are defined according to the RTF format.  
|               | - SGML. Data fields are defined according to the SGML format.  
|               | - SpaceDelimited. Data fields are separated by spaces.  
|               | - TabDelimited. Data fields are separated by tabs.  
|               | For more information, see “Delimiters Component Reference” on page 86. Default is DelimiterHierarchy. |
| name          | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| pre_processor | Defines a format preprocessor that processes the input after any document processor that you defined for the pre_processor property of the example_source. You can choose one of the following options:  
|               | - HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character. It is not restricted to HTML documents.  
|               | - RtfProcessor. Normalizes RTF files. Default is empty. |
| remark        | A user-defined comment that describes the purpose or action of the component. |

**Example**

A source document has the following structure:

```plaintext
Ron  Lehrer    & & 547320876:27
Evelyn Kern    & & 9875424: 53
```

Each line of the document is a record containing a person's name, ID number, and age. The fields are separated by the symbols & & and ;. The fields contain multiple space characters at random locations.

One way to parse this document is by using **CustomFormat**. In the delimiters property of the format, assign a **DelimiterHierarchy** containing the symbols:

```plaintext
newline
& &
; 
```

In the default_transformers property, assign the **HtmlProcessor**, which removes the extra spaces from the output.

**HtmlFormat**

The **HtmlFormat** format defines the format of HTML files.
The following table describes the properties of the **HtmlFormat** format:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| default_transformers | Defines a list of transformers that the parser applies to the output of each content anchor. Default is the following list of transformers:  
- RemoveTags. Removes HTML tags.  
- HtmlEntitiesToASCII. Converts HTML entities to their ASCII equivalents.  
- HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character.  
- RemoveMarginSpace. Removes leading and trailing space. |
| delimiters        | Defines the structure of information in the document. You can choose one of the following options:  
- CommaDelimited. Data fields are separated by commas.  
- DelimiterHierarchy. Data fields are separated or surrounded by text characters.  
- HL7. Data fields are separated as defined in the HL7 standard.  
- Positional. Data fields are defined by the number of characters between them.  
- PostScript. Data fields are defined according to the PostScript format.  
- RTF. Data fields are defined according to the RTF format.  
- SGML. Data fields are defined according to the SGML format.  
- SpaceDelimited. Data fields are separated by spaces.  
- TabDelimited. Data fields are separated by tabs.  
For more information, see “Delimiters Component Reference” on page 86. Default is SGML. |
| name              | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| pre_processor     | Defines a format preprocessor that processes the input after any document processor that you defined for the pre_processor property of the example_source. You can choose one of the following options:  
- HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character. It is not restricted to HTML documents.  
| remark            | A user-defined comment that describes the purpose or action of the component. |

**RtfFormat**

The **RtfFormat** format defines the format of RTF files.
The following table describes the properties of the **RtfFormat** format:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_transformers</td>
<td>Defines a list of transformers that the parser applies to the output of each content anchor. Default is the following list of transformers:</td>
</tr>
<tr>
<td></td>
<td>- RtfToASCII. Removes RTF control words from the output.</td>
</tr>
<tr>
<td></td>
<td>- RemoveRtfFormatting. Removes RTF formatting instructions from the text.</td>
</tr>
<tr>
<td></td>
<td>- HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character.</td>
</tr>
<tr>
<td></td>
<td>- RemoveMarginSpace. Removes leading and trailing space.</td>
</tr>
<tr>
<td>delimiters</td>
<td>Defines the structure of information in the document. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- CommaDelimited. Data fields are separated by commas.</td>
</tr>
<tr>
<td></td>
<td>- DelimiterHierarchy. Data fields are separated or surrounded by text characters.</td>
</tr>
<tr>
<td></td>
<td>- HL7. Data fields are separated as defined in the HL7 standard.</td>
</tr>
<tr>
<td></td>
<td>- Positional. Data fields are defined by the number of characters between them.</td>
</tr>
<tr>
<td></td>
<td>- PostScript. Data fields are defined according to the PostScript format.</td>
</tr>
<tr>
<td></td>
<td>- RTF. Data fields are defined according to the RTF format.</td>
</tr>
<tr>
<td></td>
<td>- SGML. Data fields are defined according to the SGML format.</td>
</tr>
<tr>
<td></td>
<td>- SpaceDelimited. Data fields are separated by spaces.</td>
</tr>
<tr>
<td></td>
<td>- TabDelimited. Data fields are separated by tabs.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “Delimiters Component Reference” on page 86.</td>
</tr>
<tr>
<td></td>
<td>Default is RTF.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>pre_processor</td>
<td>Defines a format preprocessor that processes the input after any document processor that you defined for the pre_processor property of the example_source. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character. It is not restricted to HTML documents.</td>
</tr>
<tr>
<td></td>
<td>- RtfProcessor. Normalizes RTF files.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

**TextFormat**

The **TextFormat** format defines the format of text files.

Use this format in combination with a document processor to process other types of documents. For example, you can use it with the **PdfToTxt_4** document processor to process PDF documents.
The following table describes the properties of the **TextFormat** format:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_transformers</td>
<td>Defines a list of transformers that the parser applies to the output of each content anchor.  Default is the following list of transformers:  - HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character.  - RemoveMarginSpace. Removes leading and trailing space.</td>
</tr>
<tr>
<td>delimiters</td>
<td>Defines the structure of information in the document. You can choose one of the following options:  - CommaDelimited. Data fields are separated by commas.  - DelimiterHierarchy. Data fields are separated or surrounded by text characters.  - HL7. Data fields are separated as defined in the HL7 standard.  - Positional. Data fields are defined by the number of characters between them.  - PostScript. Data fields are defined according to the PostScript format.  - RTF. Data fields are defined according to the RTF format.  - SGML. Data fields are defined according to the SGML format.  - SpaceDelimited. Data fields are separated by spaces.  - TabDelimited. Data fields are separated by tabs.  For more information, see “Delimiters Component Reference” on page 86. Default is DelimiterHierarchy.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>pre_processor</td>
<td>Defines a format preprocessor that processes the input after any document processor that you defined for the pre_processor property of the example_source. You can choose one of the following options:  - HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character. It is not restricted to HTML documents.  - RtfProcessor. Normalizes RTF files. Default is empty.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

**XmlFormat**

The **XmlFormat** format defines the format of XML files.

The parser treats the XML input document as ordinary text. You can define delimiters, anchors, and other components just as you do for a regular text document.
The following table describes the properties of the XmlFormat format:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_transformers</td>
<td>Defines a list of transformers that the parser applies to the output of each content anchor. Default is the following list of transformers:</td>
</tr>
<tr>
<td></td>
<td>- RemoveTags. Removes XML tags from the output.</td>
</tr>
<tr>
<td></td>
<td>- HtmlEntitiesToASCII. Converts XML entities to their ASCII equivalents.</td>
</tr>
<tr>
<td></td>
<td>- HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character.</td>
</tr>
<tr>
<td></td>
<td>- RemoveMarginSpace. Removes leading and trailing space.</td>
</tr>
<tr>
<td>delimiters</td>
<td>Defines the structure of information in the document. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- CommaDelimited. Data fields are separated by commas.</td>
</tr>
<tr>
<td></td>
<td>- DelimiterHierarchy. Data fields are separated or surrounded by text characters.</td>
</tr>
<tr>
<td></td>
<td>- HL7. Data fields are separated as defined in the HL7 standard.</td>
</tr>
<tr>
<td></td>
<td>- Positional. Data fields are defined by the number of characters between them.</td>
</tr>
<tr>
<td></td>
<td>- PostScript. Data fields are defined according to the PostScript format.</td>
</tr>
<tr>
<td></td>
<td>- RTF. Data fields are defined according to the RTF format.</td>
</tr>
<tr>
<td></td>
<td>- SGML. Data fields are defined according to the SGML format.</td>
</tr>
<tr>
<td></td>
<td>- SpaceDelimited. Data fields are separated by spaces.</td>
</tr>
<tr>
<td></td>
<td>- TabDelimited. Data fields are separated by tabs.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “Delimiters Component Reference” on page 86.</td>
</tr>
<tr>
<td></td>
<td>Default is SGML.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>pre_processor</td>
<td>Defines a format preprocessor that processes the input after any document processor that you defined for the pre_processor property of the example_source.</td>
</tr>
<tr>
<td></td>
<td>You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- HtmlProcessor. Converts all combinations of tab, space, or newline to a single space character. It is not restricted to HTML documents.</td>
</tr>
<tr>
<td></td>
<td>- RtfProcessor. Normalizes RTF files.</td>
</tr>
<tr>
<td></td>
<td>Default is HtmlProcessor.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

**Delimiters Component Reference**

A delimiters component defines a hierarchy of characters or strings that organize the information in a document, such as newlines, spaces, tabs, commas, or vertical bars. You can also use a wildcard pattern to define the delimiters.

The delimiter concept is applicable both to rigidly structured documents that use predefined delimiter characters to separate the data fields, and to loosely structured text or HTML documents that are delimited by newlines and syntactic markup. The delimiter concept also encompasses positionally-structured data, where the fields are located at fixed offsets from one another.

The parser uses the delimiters to determine the search criteria of Content anchors configured with the LearnByExample option.

For example, suppose you configure a format with the TabDelimited delimiters component. This defines a hierarchy using the following characters as delimiters:

Newline
Tab
You might define a Content anchor that is located two tab characters after the preceding Marker anchor in the example source, like this:

```
MARKER<tab>abc<tab>CONTENT
```

When a parser processes a source document, it searches for the Content two tabs after the Marker.

In a second example, you might define a Content anchor that is located three newlines and one tab after a Marker anchor, in the example source.

```
MARKER
abc<tab>de
fghi<tab>ijkl<tab>mnop
pqrst<tab>CONTENT
```

Within the intermediate lines, the tabs are not counted because the newlines are higher in the hierarchy.

Many of the delimiters components, such as TabDelimited or CommaDelimited, display a predefined hierarchy of delimiters, which you can edit as required.

The DelimiterHierarchy component does not have a predefined hierarchy. You can insert whatever delimiters you need.

**CommaDelimited**

The CommaDelimited delimiters component defines the following delimiter hierarchy:

```
Newline
Comma
```

Use CommaDelimited when each line of a text file contains a record and each record contains data fields separated by commas.

You can add additional delimiters or edit the predefined hierarchy. Use the same procedure that you use to edit the DelimiterHierarchy component.

**Example**

In the source document, a Content anchor follows a Marker anchor by two lines. In the third line, there are three commas, plus any other text, before the Content anchor:

```
MARKER
abcdef, ghij
abc, def,ghi,CONTENT
```

If you assign the CommaDelimited component, the parser learns from the example source that the Content anchor always follows the Marker by two newlines and three commas. In another source document, the parser will successfully find the following Content anchor:

```
MARKER
xyz, uvw, rst
,,CONTENT
```

**Delimiter**

TheDelimiter subcomponent defines a delimiter character or string that separates anchors. You can add Delimiter subcomponents within a delimiter hierarchy.
The following table describes the properties of the Delimiter subcomponent:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| search   | Defines the delimiter. You can choose one of the following options:  
  - NewlineSearch. The delimiter is a newline.  
  - PatternSearch. The delimiter is defined by a regular expression.  
  - TextSearch. The delimiter is an explicit string or a string that you retrieve dynamically from the source document.  
  For more information, see “Searcher Component Reference” on page 144. |

**Example**

The TabDelimited component contains two Delimiter subcomponents. The first uses NewlineSearch to define the newline character as a delimiter. The second uses TextSearch to define the tab character as a delimiter. The tab is graphically represented as a « character.

The SpaceDelimited component also contains two Delimiter subcomponents. The first is identical to that of TabDelimited. The second uses a PatternSearch to define any string of one or more spaces as a delimiter. The regular expression \[ ]+ means “one or more space characters.” Note the space between the square brackets.

### DelimiterHierarchy

The DelimiterHierarchy delimiters component allows you to define a custom delimiter hierarchy. Under DelimiterHierarchy, you can nest any number of Delimiter or EnclosingDelimiters components.

**Example**

In the example source document, suppose that the anchors are separated by commas and surrounded by brackets, like this:

```
MARKER,,[CONTENT]
```

You might define a DelimiterHierarchy that contains:

```
comma //defined as a Delimiter component
[ ] //defined as an EnclosingDelimiters component
```

From this example, the parser learns that the Content anchor follows the Marker by two commas and is surrounded by brackets. In another source document, the parser will find the following Content anchor:

```
MARKER,abc,def[CONTENT]
```

**Online Sample**

For an online sample, see samples\Projects\EDI\EDI.cmw. The sample uses a DelimiterHierarchy to define the newline and asterisk (*) characters as delimiters, in an EDI source document.

### EnclosingDelimiters

The EnclosingDelimiters subcomponent defines a pair of delimiter characters or strings, which surround anchors. You can add EnclosingDelimiters subcomponents under a delimiter hierarchy.

You can use this component to define the curly brace ({{}) delimiters that surround blocks of C program code.
The following table describes the properties of the `EnclosingDelimiters` subcomponent:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opening</td>
<td>Defines the opening delimiter.</td>
</tr>
<tr>
<td>closing</td>
<td>Defines the closing delimiter.</td>
</tr>
<tr>
<td>escape_sequence</td>
<td>Defines a prefix that causes the parser to ignore an instance of the opening or closing delimiter in the source document.</td>
</tr>
</tbody>
</table>

**HL7**

The HL7 delimiters component defines the following hierarchy of delimiters for parsing HL7 messages:

- `newline`
- `vertical bar (|)`
- `caret (^)` or `tab`

You can add additional delimiters or edit the predefined hierarchy. The procedure is the same as for the `DelimiterHierarchy` component.

The HL7 messaging standard permits a message to define its own delimiters. You can parse the delimiter declaration of an HL7 message and create a dynamic delimiter definition in the following way:

1. Use `Content` anchors to retrieve the delimiter characters from the HL7 message header. Store the characters in variables.
2. Add `Delimiter` components under the `HL7` component.
3. To each `Delimiter` component, assign `TextSearch`.
4. Under the `TextSearch` component, assign one of the variables to the `text` property.

**Positional**

The Positional delimiters component causes the parser to find content anchors by counting the characters from the beginning of the search scope. For more information about search scope, see “Anchors Overview” on page 106.

**Example**

In the example source document, suppose that a `Content` anchor follows a `Marker` anchor by five characters, possibly including spaces, tabs, and so forth:

```
MARKERab cdCONTENTefg
```

If you assign the `Positional` component, the parser learns from the example source that the `Content` anchor always follows the `Marker` by five characters, and that it is seven characters long. In another source document, the parser will successfully find the following `Content` anchor:

```
MARKERd<tab>cbaCONTENTzy,xwv
```

**Using Positional Parsing Together with Delimiters**

You cannot add delimiters to the `Positional` component.

Sometimes, you might want to define a parser that uses delimiters to locate some anchors, and uses a positional definition for other anchors. To do this, select one of the other delimiters components. Do not use `Positional`. To define the location of an anchor positionally, you can assign the `OffsetSearch` option in the anchor properties.
PostScript

The PostScript delimiters component defines a delimiter hierarchy that is used for parsing Adobe PostScript documents.

You cannot edit the delimiter hierarchy of the PostScript component.

RTF

The RTF delimiters component defines a delimiter hierarchy for parsing RTF documents.

You cannot edit the delimiter hierarchy of the RTF component.

SGML

The SGML delimiters component defines a delimiter hierarchy for parsing SGML, HTML, and XML documents.

You cannot edit the delimiter hierarchy of the SGML component.

SpaceDelimited

The SpaceDelimited delimiters component defines the following delimiter hierarchy:

```
Newline
String of one or more space characters
```

SpaceDelimited is used when each line of a text file contains a record and each record contains data fields separated by spaces.

You can add additional delimiters or edit the predefined hierarchy. The procedure is the same as for the DelimiterHierarchy component.

Example

In the example source document, suppose that a Content anchor follows a Marker anchor by two lines. In the third line, there are two space characters and one string containing multiple spaces before the Content anchor, like this:

```
MARKER
abcdef
abc def ghi CONTENT
```

If you assign the SpaceDelimited component, the parser learns from the example source that the Content anchor always follows the Marker by two lines and three strings of spaces. In another source document, the parser will successfully find the following Content anchor:

```
MARKER
ghi xyz def abc CONTENT
```

TabDelimited

The TabDelimited delimiters component defines the following delimiter hierarchy:

```
Newline
Tab
```

TabDelimited is used when each line of a text file contains a record and each record contains data fields separated by tabs.

You can add additional delimiters or edit the predefined hierarchy. The procedure is the same as for the DelimiterHierarchy component.
Example

In the example source document, suppose that a **Content** anchor follows a **Marker** anchor by two lines. In the third line, there are three tab characters, plus any other text, before the **Content** anchor, like this:

```markdown
MARKER
abcdef
abc<tab> de,f<tab>ghi<tab>CONTENT
```

If you assign the **TabDelimited** component, the parser learns from the example source that the **Content** anchor always follows the **Marker** by two lines and three tabs. In another source document, the parser will successfully find the following **Content** anchor:

```markdown
MARKER
xyz
<tab><tab><tab>CONTENT
```

Format Preprocessor Component Reference

The following list describes the differences between format preprocessors and document processors:

- You can assign a document processor to the `pre_processor` property of an input port, located under the `example_source` or `sources_to_extract` property of a parser. You can assign a format preprocessor only to the `pre_processor` property of a format.

- A document processor runs on the source document before it performs any other operations.

- A format preprocessor runs on the text before it searches for anchors. The output of the format preprocessor is not displayed.

For more information, see “Document Processors Overview” on page 66.

**HtmlProcessor**

The **HtmlProcessor** format preprocessor, which also functions as a transformer, normalizes whitespace according to HTML conventions. It reduces any combination of tabs, line breaks, and space characters to a single space character.

Use this preprocessor to normalize whitespace in any type of text. It is not restricted to HTML documents.

**RtfProcessor**

The **RtfProcessor** format preprocessor normalizes the code of RTF files.
CHAPTER 10

Data Holders

This chapter includes the following topics:
- Data Holders Overview, 92
- XML Schemas, 92
- Using a Schema to Map Anchors, 96
- Generating Valid XML, 97
- Variables, 99
- Variable Component Reference, 102
- Multiple-Occurrence Data Holders, 103

Data Holders Overview

A data holder is an object that has one of the following types:
- An XML element
- An XML attribute
- A variable

XML elements and attributes are typically used for permanent storage. A parser, for example, stores its output in data holders of these types.

Variables are used for temporary storage. For example, a parser can store data that it extracts from a source document in a variable. It can process the data further before creating the output.

Every data holder has a data type. In the case of elements and attributes, the data holders are defined in an XML schema that you must supply. Variables are defined in an internal schema, which you can customize by adding user-defined variables.

XML Schemas

When you create a parser, serializer, or mapper, you must supply one or more XML schemas that define the structure of the XML. The schema defines the elements and attributes that the transformation can use.

You must add the schema to your project. You can then map the content of a document to elements and attributes that are defined in the schema.
About Schemas

Scripts require XML schemas that define the input or output XML structures. The schemas must comply with the XML Schema recommendation, published by the World Wide Web Consortium. Schema files typically have an *.xsd file extension.

The following is a simple example of a schema:

```xml
<?xml version="1.0" encoding="Windows-1252"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" minOccurs="0"/>
        <xs:element name="First" minOccurs="0" type="xs:string"/>
        <xs:element name="Last" minOccurs="0" type="xs:string"/>
      </xs:sequence>
      <xs:simpleType>
        <xs:string/>
      </xs:simpleType>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The schema defines the elements and attributes that can occur in an XML document. The syntax lets a schema author specify the hierarchy and sequence of elements, whether the elements are mandatory or required, their data types, their possible values, and many other features.

The above sample schema defines an XML structure such as the following:

```xml
<Person gender="M">
  <Name>
    <First>Ron</First>
    <Last>Lehrer</Last>
  </Name>
  <ID>547329876</ID>
  <Age>27</Age>
</Person>
```

If you trace through the schema, you can observe the correspondence between definitions such as `<xs:element name="Person">` or `<xs:attribute name="gender" type="xs:string"/>` and the elements and attributes of the XML.

The elements and attributes have data types, such as `xs:string`. An element that contains nested elements or attributes has a type of `xs:complexType`. The elements have many other properties, such as their required `sequence` and the minimum number of times that they must occur in an XML document, `minOccurs`.

For more information about the schema syntax, see the following websites:

<table>
<thead>
<tr>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.w3.org">http://www.w3.org</a></td>
<td>The website of the World Wide Web Consortium, which created and maintains</td>
</tr>
<tr>
<td></td>
<td>the XML Schema recommendation.</td>
</tr>
<tr>
<td><a href="http://www.w3schools.com">http://www.w3schools.com</a></td>
<td>See this site for a tutorial introduction to XML Schema.</td>
</tr>
</tbody>
</table>
How to Create Schemas

You can create a schema in any XML schema editor. Typically, a schema editor has a user-friendly interface that helps you create and edit schemas even if you do not know the schema syntax. Some editors also let you convert an existing DTD or XDR schema to the XML schema (*.xsd) format, or to create a schema from a sample XML file.

If you know the schema syntax, you can also edit schemas in a text editor such as Notepad.

Schema Encoding

Save the schema in one of the supported input encodings.

The schema encoding must be compatible with the working encoding that you use in the IntelliScript editor. This means that:

- The schema encoding is identical to the working encoding,
- or
- Every character in the schema has an equivalent in the working encoding. For example, if the schema uses the UTF-8 encoding, and the working encoding is Windows-1252, the schema must not contain Unicode characters that have no Windows-1252 equivalent.

When you add a schema from an external location to a project, the script translates the project copy of the schema to the working encoding.

Included Schema Files

A schema can reference additional schema files. This feature lets you maintain a large schema in a modular fashion.

Namespaces

If you plan to work with XML namespaces, assign the targetNamespace attribute of the schema. You can edit the alias that is assigned to the namespace.

You cannot add two schemas that use an empty alias for different namespaces, or two schemas that use the same alias for different namespaces.

Mixed Content

Elements can contain both character data and nested elements. You can use the mixed attribute in a schema.

The script distinguishes between character data before and after each element. For more information, see “Mapping Mixed Content” on page 96.
### Unsupported Schema Features

The current version does not support certain uses of schema features. The following table lists the known limitations:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniqueness constraints</td>
<td>The <code>unique</code>, <code>key</code>, and <code>keyref</code> elements are ignored. The event log includes a warning.</td>
</tr>
<tr>
<td>Default values for elements of mixed type</td>
<td>The script ignores the default. The event log includes a warning.</td>
</tr>
<tr>
<td>Default data type</td>
<td>If the type of an element is undefined, the script processes it as <code>xs:string</code>. The event log includes a warning.</td>
</tr>
<tr>
<td></td>
<td>You can change the default to <code>xs:anyType</code>.</td>
</tr>
<tr>
<td>Regular expressions</td>
<td>There are minor discrepancies between the regular expression processor and the schema standard.</td>
</tr>
<tr>
<td>Sequence defining multiple elements having the same name</td>
<td>If an <code>xs:sequence</code> contains multiple <code>xs:element</code> definitions having the same name, the script processes only the first <code>xs:element</code>. The event log includes a warning.</td>
</tr>
<tr>
<td></td>
<td>To resolve the problem, wrap each <code>xs:element</code> in an independent <code>xs:sequence</code>.</td>
</tr>
<tr>
<td>Minimum and maximum dates</td>
<td>If a facet defines a minimum or maximum value for an <code>xs:date</code> element, the transformation fails.</td>
</tr>
<tr>
<td>Lax or skip validation options</td>
<td>In an <code>xs:any</code> or <code>xs:anyAttribute</code> element, the script ignores a <code>processContents</code> value of <code>lax</code> or <code>skip</code>. It behaves as if the value were <code>strict</code>.</td>
</tr>
<tr>
<td>Substitution group</td>
<td>The script permits a <code>substitutionGroup</code>, even if a <code>block</code> or <code>blockDefault</code> attribute forbids substitutions.</td>
</tr>
<tr>
<td>XSI type</td>
<td>The script permits an <code>xsi:type</code> attribute even if a <code>block</code> attribute of forbids it.</td>
</tr>
<tr>
<td>Built-in types</td>
<td>Some built-in types do not have correct patterns, for example, when they include characters above ASCII 127.</td>
</tr>
<tr>
<td>Substitution group without a type</td>
<td>The script sometimes fails when a substitution group does not have a type.</td>
</tr>
<tr>
<td>Empty namespace</td>
<td>When the namespace is empty, the script adds an alias to all elements in the source file, but the alias does not appear on the <code>Locator</code> and the <code>Locator</code> fails.</td>
</tr>
<tr>
<td>List</td>
<td>The script reads a space-separated <code>xs:list</code> as a single item, which might fail if its length exceeds the stated limit for individual items in the list.</td>
</tr>
<tr>
<td>Floats and doubles</td>
<td><code>xs:float</code> and <code>xs:double</code> do not accept valid values of <code>INF</code>, <code>-INF</code>, or <code>NaN</code>.</td>
</tr>
<tr>
<td>Element with both fixed and mixed attributes</td>
<td>The script does not read all parts of an element that has both fixed and mixed attributes.</td>
</tr>
<tr>
<td><code>max_occurs=0</code></td>
<td>The script creates output even when <code>max_occurs=0</code>.</td>
</tr>
</tbody>
</table>
### Precision of Numerical Data

The script stores `xs:decimal` and `xs:float` data as strings, preserving the precision of the data.

In calculations, the script converts decimal and float data to double-precision floating point, and it rounds the result to 15 decimal digits. This means that decimal data may lose some precision. For example, the result of `xs:decimal 5.28 * 1` may be displayed as 5.28000000000001.

The script normalizes `xs:decimal` values. For example, it stores 0004 as 4, -0 as 0, and 1.200 as 1.2.

### Using a Schema to Map Anchors

When you define a parser, you must map `Content` anchors to output data holders. When you define a serializer, you must map input data holders to `ContentSerializer` serialization anchors.

### IntelliScript Representation of Data Holders

In the script, data holders are identified by a modified XPath expression, such as:

\[
\text{data\_holder} = /\text{Person/*/s/Name/*/s/First}
\]

To change this value, select the `data\_holder` property and press `ENTER`. This opens a `Choose XPath` dialog box, where you can select the new value.

The XPath syntax is slightly different from the standard XPath syntax, which is `Person/Name/First`. The script inserts `*s`, `*c`, and `*a`, which refer to the schema terms `sequence`, `choice`, and `all`. The modifications resolve ambiguities when the script uses the schema to help construct XML output.

### Mapping Mixed Content

If the schema supports mixed content, each element has `before` and `after` data holders. For example, consider the following mixed content:

```xml
<Deal>
    We are pleased to offer you a price of
    <Price>34</Price>
    dollars. This is a special price for
    <Partner>
        <Name>Acme Gizmos, Inc.</Name>
        <ID>98765</ID>
    </Partner>
    valid only until December 31.
</Deal>
```

This structure contains data holders in the following locations:

- Immediately after the `<Deal>` tag, before any of the sub-elements.
• Before the Price element
• The Price element
• After the Price element
• Before the Partner element
• The Partner/Name and Partner/ID elements
• After the Partner element
• Immediately before the </Deal> tag, after all the sub-elements.

You can map the text "We are pleased to offer you a price of" to the data holder before the Price element. You can map "dollars. " to the data holder after Price, and "This is a special price for " to the data holder before Partner.

The following example shows mixed content:

```xml
data_holder = /Deal/*/s/Price/$text_before
```

Mapping XSI Types

A schema can define derived data types that can be used in place of a base type. In such cases, an XML document can define the actual data type of an element by specifying an xsi:type attribute.

For example, a schema defines a Person element having a type PersonT1 and containing string content. It defines a type called PersonT2 that extends PersonT1 by adding an Id attribute. The following are valid Person elements:

```xml
<!-- base type PersonT1 -->
<Person>Ron Lehrer</Person>

<!-- derived type PersonT2 -->
<Person Id="547329876" xsi:type="PersonT2">Ron Lehrer</Person>
```

The script interprets xsi:type attributes in input XML documents. It adds xsi:type attributes where necessary to output XML documents.

Select the appropriate type according to the data that the transformation processes. For example, if you want a Content anchor to store data in a Person element having type PersonT2, select xsi:type=PersonT2. The selection appears in the script as follows:

```xml
data_holder=/Person/*c/xsi:type=PersonT2
```

In cases where the content might require either a PersonT1 or PersonT2 data holder, you can configure an Alternatives anchor that contains two Content anchors. One of the Content anchors is mapped to PersonT1, and the other to PersonT2. For more information, see “Alternatives” on page 118.

If you map a data holder to the unqualified element Person, the data holder defaults to the base type PersonT1. Thus the following mappings are equivalent:

```xml
data_holder=/Person
data_holder=/Person/*c/xsi:type=PersonT1
```

Generating Valid XML

The script generates XML that is valid according to the output schema that you have defined.

The schema is used as a guide while the XML is being generated. The schema is applied during the generation, and not afterwards. This approach helps transformations to succeed. It ensures the validity continually as the transformation proceeds.
Role of Schemas in Parsing

This section explains some of the ways in which a parser uses the schema to ensure that it outputs valid XML.

Sequence of Elements

When the script runs a parser, it organizes the output in the sequence that is required by the schema.

For example, a schema may require that a LastName element precede a FirstName element. The script creates the output in the locations defined by the schema, even if the anchors that produce the output are defined in the opposite sequence.

Number of Occurrences

A parser may attempt to insert multiple instances of an element in the output XML. The script uses the schema to determine whether to append new instances or overwrite existing elements. The parser deletes any excess elements beyond those that the schema permits, and it writes warnings in the event log.

In another example, suppose the schema defines an element without specifying a minOccurs or maxOccurs attribute. The default minOccurs and maxOccurs values are 1, which means that the element must occur exactly once in the parser output. If the element is missing from the output, the parser can add it.

For more information, see “Multiple-Occurrence Data Holders” on page 103.

Missing or Empty Elements

In the Data Processor transformation settings, you can configure whether a parser inserts empty elements to comply with a schema.

Data Types

The script ensures that the text it stores in a data holder has the required data type. For example, if a Content anchor retrieves the string "oranges 5 for a dollar", and the type of the data holder is xs:integer, the anchor stores only the integer 5 in the data holder.

For more information, see “Using Data Types to Narrow the Search Criteria” on page 116.

Role of Schemas in Serialization and Mapping

A serializer or mapper checks that its input is valid according to the XML schema. There are two validation modes:

- Partial validation. Some deviations are allowed between the XML source document and the schema. Default.
- Strict validation. The XML source document must conform strictly to its schema.

To define the validation level, assign the validate_source_document property of the Serializer or Mapper component.

If you use the strict mode, a validation error causes the serializer or mapper to fail. The Events view displays the errors.

If you use the partial mode, the transformation might proceed despite certain validation errors. For example, if there are more occurrences of an element than the schema permits, a serializer typically ignores the excess elements and processes the valid ones, and it writes a warning in the event log. Similarly, it might ignore an element containing an invalid data type.

The script uses the Xerces C XML parser, version 3.1, to perform validation.
Variables

Variables are temporary data holders that you can use in place of XML elements or attributes. Variables are useful if you need to store a value temporarily during the operation of a transformation, and you do not need to output the value in the XML.

For example, suppose you want a parser to read two Content anchors and concatenate their values. You might map each Content anchor to a user-defined variable. You can then use an action to concatenate the variables and output the result to an XML element.

The script also uses pre-defined system variables to store information that is needed in certain operations.

Creating a User-Defined Variable

1. Add a Variable component at the global level of the script.
2. Enter a name for the variable, and then press ENTER.
3. Select the data type that the variable can store.
   You can select a standard type such as xs:string or xs:integer, or a global type defined in a schema referenced in the project.

System Variables

The following paragraphs describe the system variables and the ways in which they are used.

Variables Used to Access Source Documents

Several of the system variables store data that actions can use to access source documents. For example, the RunParser action can use VarLinkURL, which contains a file path.

The following variable is used in the XmlDocument processor:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarServiceInfo &gt; ServiceLocation</td>
<td>The directory path of the script or service that is currently running.</td>
</tr>
</tbody>
</table>
Read-Only Access Variables

The following variables are read-only. A transformation can use them to visit a source document more than one time.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarRequestedURL</td>
<td>The path of the source document that a parser is processing.</td>
</tr>
<tr>
<td>VarCurrentURL</td>
<td>The path of the current file that a parser is processing. Usually, this is the same as VarRequestedURL. If the parser is configured with certain preprocessors, VarCurrentURL might point to a temporary file rather than the original source document. VarRequestedURL always points to the source document.</td>
</tr>
<tr>
<td>VarCurrentPost</td>
<td>The form data that a parser submitted to retrieve the current page.</td>
</tr>
</tbody>
</table>

Read-Only System Time Variables

VarSystem is a read-only variable that returns system information. It is a structure containing several nested variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarSystem &gt; ExecStartTime &gt; Year</td>
<td>Year when the transformation began execution</td>
</tr>
<tr>
<td>VarSystem &gt; ExecStartTime &gt; Month</td>
<td>Numerical month</td>
</tr>
<tr>
<td>VarSystem &gt; ExecStartTime &gt; MonthName</td>
<td>Name of month</td>
</tr>
<tr>
<td>VarSystem &gt; ExecStartTime &gt; Day</td>
<td>Day of month</td>
</tr>
<tr>
<td>VarSystem &gt; ExecStartTime &gt; DayName</td>
<td>Day of week</td>
</tr>
<tr>
<td>VarSystem &gt; ExecStartTime &gt; Hour</td>
<td>Hour</td>
</tr>
<tr>
<td>VarSystem &gt; ExecStartTime &gt; Minute</td>
<td>Minute</td>
</tr>
<tr>
<td>VarSystem &gt; ExecStartTime &gt; Second</td>
<td>Second</td>
</tr>
<tr>
<td>VarSystem &gt; ExecStartTime &gt; Millisecond</td>
<td>Millisecond</td>
</tr>
</tbody>
</table>

You can use VarSystem to insert a timestamp in the output of a transformation.

Variables Used for Failure Handling

VarLastFailure stores the most recent component failure that occurred in a transformation. For example, it might record an instance of a Marker anchor that failed to find the marker text. You can configure a component to write VarLastFailure to a user log when a failure occurs. For more information, see “Failure Handling” on page 289.

Note: When you use VarLastFailure, the service runs in special mode, which requires about three times more CPU time.

VarServiceInfo stores the service name, directory location of the user log, and the file name of the user log.
**VarLastFailure** and **VarServiceInfo** are structures containing the following nested variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarLastFailure &gt; InternalId</td>
<td>Failure identifier</td>
</tr>
<tr>
<td>VarLastFailure &gt; Text</td>
<td>Failure description</td>
</tr>
<tr>
<td>VarLastFailure &gt; Location</td>
<td>Location of the failure in the script</td>
</tr>
<tr>
<td>VarLastFailure &gt; AnchorName</td>
<td>Name of the component that failed</td>
</tr>
<tr>
<td>VarLastFailure &gt; Data</td>
<td>Additional information about the failure</td>
</tr>
<tr>
<td>VarServiceInfo &gt; ServiceName</td>
<td>Name of the service</td>
</tr>
<tr>
<td>VarServiceInfo &gt; StandardError &gt; StandardErrorDir</td>
<td>Directory path of the user log</td>
</tr>
<tr>
<td>VarServiceInfo &gt; StandardError &gt; StandardErrorName</td>
<td>File name of the user log</td>
</tr>
</tbody>
</table>

**Variables Used for Structured Parsing**

**VarStructureDetails** keeps track of the current record that a **StructureDefinition** anchor is parsing. It contains the following nested variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarStructureDetails &gt; Name</td>
<td>The <em>name</em> property of the subelement that matches the record.</td>
</tr>
<tr>
<td>VarStructureDetails &gt; Repetitions</td>
<td>The iteration number of the record.</td>
</tr>
<tr>
<td>VarStructureDetails &gt; RecordIndex</td>
<td>The index number of the record in the overall <strong>StructureDefinition</strong> input.</td>
</tr>
<tr>
<td>VarStructureDetails &gt; RecordId</td>
<td>The record identifier. If there are multiple identifiers, the variable contains a comma-separated list.</td>
</tr>
<tr>
<td>VarStructureDetails &gt; InternalPath</td>
<td>Internal information, not for use.</td>
</tr>
</tbody>
</table>

For more information, see “StructureDefinition” on page 140.

**Variables Used in Notifications**

**VarNotificationDetails** stores information about the most recent notification that was triggered in a transformation:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarNotificationDetails &gt; Name</td>
<td>The name of the notification.</td>
</tr>
<tr>
<td>VarNotificationDetails &gt; Path</td>
<td>The XPath of the data holder to which the notification applies. For example, if a validator triggers a notification in a <strong>Content</strong> anchor, the <strong>Path</strong> is the data holder where the <strong>Content</strong> anchor stores its output.</td>
</tr>
</tbody>
</table>
### Variable Component Reference

A **Variable** component is a user-defined variable.

For more information about system variables, see “System Variables” on page 99.

#### Variable

A **Variable** component is a user-defined variable that you use in a script.

Use variables for temporary storage in the same way that you use an XML element or attribute. For example, you can map a **Content** anchor to a variable, and you can use a variable as the input of an action.

Variables appear at the global level of the script. A variable can have any data type that is defined in the schemas associated with the project, including standard types and custom types. A custom type can be either simple or complex. A complex variable is a structure containing nested fields. Initialization of complex variables is not supported. For more information, see “Initializing Variables at Runtime” on page 102.

---

<table>
<thead>
<tr>
<th>Variable Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarNotificationDetails &gt; Value</td>
<td>The input value that caused the notification. If a validator triggers the notification, the <strong>Value</strong> is the invalid input data. If a <strong>Notify</strong> action triggers the notification, you can specify the <strong>Value</strong> in the <strong>Notify</strong> configuration.</td>
</tr>
<tr>
<td>VarNotificationDetails &gt; Creator</td>
<td>The location in the script that triggered the notification.</td>
</tr>
</tbody>
</table>

For more information, see “Notifications” on page 307.
The following table describes the properties of the **Variable** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>Defines an initial value for the variable. You can initialize variables that have simple data types. Default is empty.</td>
</tr>
<tr>
<td>InitialValue</td>
<td>Defines an initial value for the variable. InitialValue has one property, value.</td>
</tr>
</tbody>
</table>
| list | Determines whether the variable is single-occurrence or multiple-occurrence variable. You can choose one of the following options:  
  - Selected. Determines a multiple-occurrence variable.  
  - Cleared. Determines a single-occurrence variable.  
  Default is cleared. For more information, see “Multiple-Occurrence Data Holders” on page 103. |
| val_type | Defines the data type that the variable can store. Legal values are defined in the schema. Default is xs:string. |
| value | Defines the initial value. |

### Multiple-Occurrence Data Holders

In a schema, you can use the `maxOccurs` attribute to set the maximum number of times that sibling elements can occur in an XML document. Likewise, you can define a variable that can occur either once or multiple times. An element or variable that can occur only once is called a single-occurrence data holder. An element or variable that can occur more than once is called a multiple-occurrence data holder.

Single- and multiple-occurrence data holders behave differently when the script stores data in them, for example, when you map `Content` anchors to a data holder.

- In a single-occurrence data holder, each assignment overwrites the preceding assignment.
- In a multiple-occurrence data holder, each assignment generates a new occurrence of the data holder.

To understand this, suppose that a schema defines an XML element called `<FirstName>`. If `maxOccurs = 1`, this is a single-occurrence data holder. If a parser maps more than one `Content` anchor to the `<FirstName>` element, the output contains only the final mapping.

Consider what would happen if you parse a source document that is a list of first names:

```xml
<FirstName>Jack</FirstName>
<FirstName>Jennie</FirstName>
<FirstName>Larissa</FirstName>
```

We assume that each name is a `Content` anchor mapped to `FirstName`. Each name overwrites the value of `FirstName`. The output contains only the mapping:

```xml
<FirstName>Larissa</FirstName>
```

Now suppose that `maxOccurs = unbounded`. This means that `FirstName` is a multiple-occurrence data holder. If you map multiple `Content` anchors to the element, the parser generates a list of names. The output is:

```xml
<FirstName>Jack</FirstName>
<FirstName>Jennie</FirstName>
<FirstName>Larissa</FirstName>
```

The same principle applies to variables. If you map multiple anchors to a multiple-occurrence variable, each anchor generates a new occurrence of the variable. You can use this feature, for example, to prepare input for the `AppendListItems` and `CombineValues` actions, which concatenate the occurrences.
Note: The behavior described here assumes that the multiple-occurrence data holder has a simple data type. Under certain circumstances, if the type is complex, each anchor might not generate a new occurrence. To control this behavior, you can use a locator. For more information, see “Overview of Locators, Keys, and Indexing” on page 253.

Attributes

An XML attribute is always a single-occurrence data holder. An attribute cannot be multiple-occurrence because XML does not permit the same attribute to appear more than once in the same element.

An attribute can have a data type that is a space-separated list. The names attribute in the following element is an example:

```xml
<Countries names="USA Canada Mexico"/>
```

The script treats the attribute as a single-occurrence data holder with a list type. For more information, see "Using Data Types to Narrow the Search Criteria” on page 116.

Indexing

By default, the script accesses the instances of a multiple-occurrence data holder sequentially. You can access the instances non-sequentially by using the indexing feature. For more information, see "Overview of Locators, Keys, and Indexing” on page 253.

Destroying the Occurrences

Under certain circumstances, you might want to destroy all existing occurrences of a multiple-occurrence data holder, and start creating new occurrences from the beginning of the list. This is useful, for example, if you are parsing an iterative structure, and you want to keep only the last iteration. You can destroy the occurrences that store data from the earlier iterations.

You can achieve this effect by defining a single-occurrence data holder that contains a nested, multiple-occurrence element. When you re-use the single-occurrence data holder, the nested occurrences are destroyed.

The following scenario is a typical example.

1. Add the following schema to a project:

   ```xml
   <xs:schema xmlns:x="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
   <xs:complexType name="MyListType">
     <xs:sequence>
       <xs:element name="item" type="xs:string" maxOccurs="unbounded"/>
     </xs:sequence>
   </xs:complexType>
   </xs:schema>
   ```

   The schema defines a custom data type called MyListType. The type contains a nested, multiple-occurrence element called item.

2. Define a single-occurrence variable called MyList, which has the data type MyListType.

3. Use the variable as the target of an iterative structure.

   For more information, see “Overview of Locators, Keys, and Indexing” on page 253.

   Each iteration re-uses the single occurrence of MyList. At the start of the iteration, the nested item elements are destroyed. Anchors within the iterative structure, such as a nested RepeatingGroup, start assigning the item elements from the beginning of the list.
Online Sample

For an example of how to destroy multiple occurrences of a data holder, see the following online sample:

samples\Projects\ResetListVariable\ResetListVariable.cmw
Chapter 11

Anchors

This chapter includes the following topics:

- Anchors Overview, 106
- Mapping Content Anchors to Data Holders, 107
- Defining Anchors, 108
- Standard Anchor Properties, 110
- How a Parser Searches for Anchors, 111
- Anchor Component Reference, 117
- Searcher Component Reference, 144
- Anchor Subcomponent Reference, 148

Anchors Overview

Anchors are the components that let a parser hook into specific locations in a source document, for the purpose of finding data and storing it in data holders. An anchor is a signpost that you place in a document, indicating the position of the data.

This chapter explains the different types of anchors and how you can use them in parsers.

Marker and Content Anchors

The most commonly used anchors are called Marker and Content anchors. These anchors are often used as a pair:

- A Marker anchor labels a location in a document.
- A Content anchor retrieves text from the location.

To understand these anchors, imagine a printed questionnaire. The first line typically asks for the person's last name and first name, with each label followed by a blank space to receive the information. The printed labels Last Name and First Name are Marker anchors, and the blank spaces are Content anchors. The anchors provide a means to home in on the data and extract it from the source document.

Other Anchor Types

In addition to Marker and Content anchors, there are many other anchor types that you can use to parse documents. For example, Group and RepeatingGroup anchors help you specify the organization of the data fields. An Alternatives anchor lets you specify multiple kinds of data that might occur at a particular location in a source document.
How Anchors and Delimiters Work Together

You can define the anchors in the example source document. The parser learns how to parse the document by examining the anchors and the delimiters that separate them. For more information about delimiters, see “Formats Overview” on page 79.

For example, suppose you have specified that your document uses a tab-delimited format. A line in the example source reads

```
First name:<tab>Ron
```

where <tab> is a tab character.

You can define `First name:` as a Marker anchor. You can define `Ron` as a Content anchor. The parser learns from these definitions that it should search a source document for the string `First name:`. It should then skip over a single tab delimiter and retrieve the text that follows the tab.

Suppose you run the parser on another source document, which contains the following text:

```
First name:<tab>Jack
```

The parser finds the anchors as above and retrieves the text `Jack`.

Now suppose that the source document reads:

```
First name:<tab>Jack<tab>Age:<tab>34
```

The parser still retrieves the text `Jack`, rather than `Jack<tab>Age:<tab>34`. This works because you have defined the tab character as a delimiter. The script understands that the Content anchor starts after the first tab and ends before the second tab. Of course, you might define additional anchors that retrieve Jack's age, which is 34.

Note: The above examples describe one possible behavior of the anchors and delimiters. The anchors have many properties that let you alter this behavior. For instance, you can define a Content anchor that ignores tabs, even in a tab-delimited format. For more information, see “How a Parser Searches for Anchors” on page 111.

Mapping Content Anchors to Data Holders

A Content anchor stores the text that it extracts from a source document in a data holder. For example, you might configure a Content anchor to store its result in an XML element called `FirstName`. If the Content anchor retrieves the text `Jack`, the parser produces the following output:

```
<FirstName>Jack</FirstName>
```

More precisely, you might specify that the anchor should store the retrieved text at the path `/Person/*s/FirstName`, which refers to an element defined in the XML schema. The actual parser output would be:

```
<Person>
  <FirstName>Jack</FirstName>
</Person>
```

On the other hand, suppose that the schema defines `FirstName` as an attribute of the Person element. You might map the Content anchor to `/Person/@FirstName`. The output would be:

```
<Person FirstName="Jack" />
```

You must map to a data holder that has an appropriate data type. For example, do not map `Jack` to an XML element that has an `xs:integer` data type, or to an XML element that has a complex data type containing nested elements. For more information about this rule, see “Using Data Types to Narrow the Search Criteria” on page 116.
Mapping to Variables

You can map an anchor to a data holder that is an XML element, an XML attribute, or a variable. The variable option is useful if you want to use the data in a subsequent processing step, but you do not want to include the raw data in the parser output.

For example, suppose you want to extract several numbers from a source document and output their sum in the XML. You do not want the individual numbers in the output. You can map the Content anchors that retrieve the numbers to variables, and use a CalculateValue action to compute and output the sum.

You might also map to a variable that you use in a subsequent anchor, for example, to define a dynamic search text for a Marker anchor.

Mapping to Multiple-Occurrence Data Holders

If you map Content anchors to a single-occurrence data holder, each assignment of the data holder overwrites the previous assignment.

If you map to a multiple-occurrence data holder, each assignment generates a new occurrence of the data holder. For example, if each Content anchor retrieves a person's name, the output is a list of names:

```xml
<FirstName>Jack</FirstName>
<FirstName>Jennie</FirstName>
<FirstName>Larissa</FirstName>
```

For more information, see “Multiple-Occurrence Data Holders” on page 103.

Mapping to Mixed-Content Elements

The term mixed content refers to an XML element that contains both character data and nested elements. If the schema permits an element to have mixed content, the Schema view displays before and after data holders for the elements. This lets you map a Content anchor to character data that is located before or after a particular nested element. For more information, see “Mapping Mixed Content” on page 96.

Defining Anchors

When you define a Parser component, you must add a sequence of anchors. The parser operates by searching for the anchors in the source document and by running the operations that you have configured the anchors to perform.

Where to Define Anchors

In the script, the anchors are nested within a Parser.

If you press ENTER at the indicated location, the IntelliScript editor displays a drop-down list that includes the anchors and other components that you can add.

After you add the anchors, the Developer tool highlights the anchors in the example source.

Some types of anchors can contain nested anchors. For example, you can nest anchors within an Alternatives, Group, or RepeatingGroup anchor.

Sequence of Anchors

The sequence of the anchors should be the sequence of text in the source document.
For example, suppose that the source document is:

First Name: Ron
Last Name: Lehrer

Assuming that you define First Name and Last Name as Marker anchors, and that you define Ron and Lehrer as Content anchors, the required sequence of anchors in the parser configuration is:

<table>
<thead>
<tr>
<th>Anchor</th>
<th>Text in the Source Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marker</td>
<td>First Name</td>
</tr>
<tr>
<td>Content</td>
<td>Ron</td>
</tr>
<tr>
<td>Marker</td>
<td>Last Name</td>
</tr>
<tr>
<td>Content</td>
<td>Lehrer</td>
</tr>
</tbody>
</table>

Exception: Variable Source Sequence

Some source documents may have a variable sequence. For example, suppose that the source document may have either of the following formats:

First Name: Ron
Last Name: Lehrer

or

Last Name: Lehrer
First Name: Ron

In such cases, you can use the marking property to change the search scope of the anchors. For more information, see "How a Parser Searches for Anchors" on page 111.

Adding a Marker or Content Anchor

You can add Marker and Content anchors by a select-and-click approach.

1. Select the anchor text in the example source file.
2. Right-click the selected text, and then click Insert Marker or Insert Content.
3. Set the anchor properties.

Defining an Anchor

You can create any type of anchor by editing the script. The procedure is identical to editing any other component.

1. At the desired anchor location, select the ellipsis (...), and then press ENTER.
2. Select or type the anchor name.
3. Press ENTER again to confirm your selection.
4. Edit the anchor properties.
The following table describes standard properties of anchors:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>direction</td>
<td>A search direction for the anchor within the search scope. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- backward. Search from the end of the search scope and finds the last instance of the anchor.</td>
</tr>
<tr>
<td></td>
<td>- forward. Search from the start of the search scope and finds the first instance of the anchor.</td>
</tr>
<tr>
<td></td>
<td>For a Marker anchor, you can modify this behavior by using the count property. For example, if direction = backward and count = 2, the script finds the second-to-last instance. Default is forward. For more information, see “How a Parser Searches for Anchors” on page 111.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>marking</td>
<td>Determines whether an anchor is used as the start of the search scope for the succeeding anchor. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- begin position. Place a reference point before the current anchor.</td>
</tr>
<tr>
<td></td>
<td>- end position. Place a reference point after the current anchor.</td>
</tr>
<tr>
<td></td>
<td>- full. Place a reference point before and after the current anchor.</td>
</tr>
<tr>
<td></td>
<td>- none. Do not create a reference point.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “How a Parser Searches for Anchors” on page 111.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>no_initial_phase</td>
<td>Determines whether the script searches for nested anchors in the main phase. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Search for nested anchors according to their individual properties.</td>
</tr>
<tr>
<td></td>
<td>- Selected. Search for nested anchors in the main phase.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notifications</td>
<td>A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307.</td>
</tr>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Take no action.</td>
</tr>
<tr>
<td></td>
<td>- CustomLog. Write to the user log.</td>
</tr>
<tr>
<td></td>
<td>- LogError. Write an error message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogInfo. Write an information message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogWarning. Write a warning message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- NotifyFailure. Send a notification.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
</tbody>
</table>
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| phase    | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

When it is not certain that an anchor exists in a source document, select the optional property. If the anchor does not exist, the Parser in which the anchor is nested continues.

If the anchor is nested within a Group anchor, the optional property prevents the Group from failing. If the anchor is in a RepeatingGroup, the property prevents an iteration of the RepeatingGroup from failing.

---

### How a Parser Searches for Anchors

To design a parser correctly, it is important that you understand how the script searches for the anchors in the parser. There are three main concepts:

- **Search phase**
- **Search scope**
- **Search criteria**

This section explains the concepts, and how you can control each of them by setting the anchor properties.

#### Search Phases

The script searches for a sequence of anchors in three phases:

- Initial
- Main
- Final

By default, all Marker anchors are in the initial phase and all Content anchors are in the main phase. This means that the script first finds the Marker anchors, and then it finds the Content anchors between them.

To understand this, consider a parser that processes the following source document:

```
First name: Ron    Last name: Lehrer
```

Suppose you have defined the anchors in the following way, with default anchor properties:

<table>
<thead>
<tr>
<th>Anchor</th>
<th>Text in the Source Document</th>
<th>Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marker</td>
<td>First name:</td>
<td>Initial</td>
</tr>
<tr>
<td>Content</td>
<td>Ron</td>
<td>Main</td>
</tr>
</tbody>
</table>
In the initial phase, the script searches for the Marker anchors:

- It searches for `First name:`.
- It searches for `Last name: at a location that follows First name:`.

In the main phase, the script searches for the Content anchors:

- It searches for the `Ron anchor at a location between First name: and Last name:`.
- It searches for the `Lehrer anchor at a location after Last name:`.

### Nested Phases

Anchors that have nested anchors, such as Group, have nested phases. For example, if a Group anchor runs in the main phase of a parser, a Marker anchor that is nested in the Group runs in a nested initial phase. The nested initial phase is part of the parser main phase, but it is before the other anchors in the Group.

Another example is a RepeatingGroup anchor, which searches for both separators and for nested anchors. In order to identify the nested anchors correctly, it searches for the separators before it searches for the nested anchors.

### Search Scope and Search Criteria

The above example of search phases illustrates the concepts of search scope and search criteria. The search scope is the portion of a document where the script searches for an anchor. The search criteria are the rules by which the script finds the anchor within the search scope.

In the initial phase, the script starts searching for the Marker anchor containing `First name:` at the beginning of the document. The search scope for this anchor is the entire document. The search criterion is that the anchor must contain the text `First name:`.

The search scope for the Last name: anchor starts at the end of First name:, and extends to the end of the document. The search criterion is that the anchor must contain the text `Last name:`.

In the main phase, the parser interpolates the Content anchors between the Marker anchors. The search scope for the Ron anchor extends from the end of the First name: anchor to the beginning of the Last name: anchor. Assuming that the parser uses a space-delimited format, the search criteria are to retrieve all the text in the search scope, after the leading space character and before the second space character.

The search scope for the Lehrer anchor is from the end of Last Name: to the end of the document. The search criteria are similar to those for the Ron anchor.
We can add this analysis to the anchor table that we presented above. The table now describes the complete method by which the parser finds the anchors.

<table>
<thead>
<tr>
<th>Anchor</th>
<th>Text in the Source Document</th>
<th>Phase</th>
<th>Search Scope</th>
<th>Search Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marker</td>
<td>First name:</td>
<td>Initial</td>
<td>Entire document</td>
<td>Text = First name:</td>
</tr>
<tr>
<td>Content</td>
<td>Ron</td>
<td>Main</td>
<td>End of First name: to start of Last name:</td>
<td>After the leading space Before the next space</td>
</tr>
<tr>
<td>Marker</td>
<td>Last name:</td>
<td>Initial</td>
<td>End of First name: to end of document</td>
<td>Text = Last name:</td>
</tr>
<tr>
<td>Content</td>
<td>Lehrer</td>
<td>Main</td>
<td>End of Last name: to end of document</td>
<td>After the leading space Before the next space</td>
</tr>
</tbody>
</table>

**Adjusting the Search Phase**

By assigning the **phase** property of an anchor, you can change the phase in which the script searches for the anchor.

Consider the following source document:

```
CONTENT<10 characters>MARKER
```

In this example, the **Marker** anchor is located 10 characters after the **Content** anchor.

By default, the script searches for the **Marker** in the initial phase, and it searches for the **Content** in the main phase. This will not work here because the script cannot find the **Marker** unless it has already found the **Content**!

The solution is to change the **phase** property of one of the anchors. You can change the **Content** to the initial phase, or the **Marker** to the main phase. In either case, the script finds the anchors.

**Adjusting the Search Scope**

There are two ways to adjust the search scope for an anchor:

- By setting the **phase** property of the anchor or the surrounding anchors
- By setting the **marking** property of the surrounding anchors

**Phase Property**

If a **Content** anchor lies between two **Marker** anchors, then by default, the search scope for the **Content** is the segment between the **Marker** anchors.

If you change all the anchors to the same phase, the search scope of the **Content** is no longer bounded by the second **Marker**. It is from the end of the first **Marker** to the end of the document.

As an example, consider the following source document:

```
Tree Fig Date<tab>October 27, 2003 (pruned)
Tree Date Palm Date April 27, 2003<tab>(planted)
```

The example assumes that the source document has a loose structure, containing varying numbers of spaces, tabs, or other symbols interspersed in the text, so we cannot easily use the spaces and tabs as delimiters. An example like this might arise in parsing word-processor documents.
We can parse this document using a RepeatingGroup anchor, which contains nested Marker and Content anchors. The Marker anchors are the strings Tree and Date. The Content anchors are everything between the Marker anchors, including the spaces and tabs.

The problem in parsing this document is in the second iteration of the RepeatingGroup, which parses the second line. If we leave the Marker anchors in the initial phase, the script incorrectly considers the first instance of the word Date to be a Marker. In the main phase, it fails to find Date Palm because the search scope is between the two Marker anchors, and there is no text between them.

A possible solution is to move the Marker for Date to the main phase, and to define the Content anchor, Date Palm, using an expression that searches for a tree name of one or two words. In the initial phase of the RepeatingGroup, the script finds the Marker for Tree. In the main phase, it finds Date Palm followed by the Marker for Date.

With the new phase setting, we have changed the search scope for the tree name. The scope is now from Tree to the end of the iteration, and the script finds Date Palm successfully.

**Marking Property**

Consider the following source-document structure:

```
MARKER
%%CONTENT A
^^CONTENT B
```

Suppose that the sequence of Content A and Content B varies among the source documents. In some documents, Content B precedes Content A.

In that case, the search criteria are:

- Content A and Content B both follow the Marker anchor.
- Content A begins with %%, and Content B begins with ^^.

By default, the search scope for Content A is from the end of the Marker to the end of the document. The search scope for Content B is from the end of Content A to the end of the document. This does not work because in some source documents, Content A and Content B are reversed.

The solution is to change the search scope for Content B. You can do this by setting the marking property of Content A. The marking property specifies where the script places the reference points that determine the start and end of the search scope.

The default setting is marking = full, which means that the script places reference points before and after each anchor. The search scope for Content B begins at the last reference point, which is the one following Content A. This leads to incorrect parsing, as we have seen.

To prevent the script from placing reference points around Content A, set the marking property of Content A to none. As a result, the search scope for Content B starts at the end of the Marker. This allows the script to find Content B, even if it precedes Content A.
The following table describes all four possible values of the marking property. The Result column assumes that you assign the marking value to Content A in the above example.

<table>
<thead>
<tr>
<th>Marking Property</th>
<th>Explanation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>The script places reference marks at the beginning and end of the current anchor. This is the default behavior.</td>
<td>The script seeks the next anchor after the end of the current anchor. Content B follows Content A.</td>
</tr>
<tr>
<td>begin position</td>
<td>The script places a reference mark only at the start of the current anchor.</td>
<td>The script seeks the next anchor after the start of the current anchor. Content B overlaps or follows Content A.</td>
</tr>
<tr>
<td>end position</td>
<td>The script places a reference mark only at the end of the current anchor.</td>
<td>The script seeks the next anchor after the end of the current anchor. Content B follows Content A.</td>
</tr>
<tr>
<td>none</td>
<td>The script does not place any reference marks at the current anchor.</td>
<td>The script seeks the next anchor after the end of the preceding anchor. Content B follows Marker, without regard to Content A.</td>
</tr>
</tbody>
</table>

Note: There are a few circumstances where you must use an anchor that marks a reference point. An example is the separator of a RepeatingGroup. If the separator does not mark, it does nothing. A warning appears if you attempt to use a non-marking anchor in a location where marking is required.

Online Samples

For an online sample of the marking property, open the project samples\Projects\Marking_Mode\Marking_Mode.cmw. The sample uses the property to alter the search scope of a Content anchor.

For another example, see samples\Projects\NonMarker\NonMarker.cmw. This sample uses the marking = none option, permitting two Content anchors to overlap. The sample also illustrates the use of direction = backward to search from the end of the scope.

Adjusting the Search Criteria

The script can search for anchors according to a large number of search criteria, for example:

- According to the delimiter locations, which the script learns from the example source
- According to a positional offset, in other words, the number of characters from a reference point
- By searching for particular text
- By searching for a pattern or regular expression
- By searching for a specified data type
- By searching for an attribute value

You can combine these search criteria in almost any way. For example, you might specify that a Content anchor begins two tabs after a Marker anchor, and that it is 10 characters long. If you do this, you are using a delimiter criterion to define the beginning of the Content anchor, and an offset criterion to define the end.

The components that perform these searches are called searcher components. For more information, see “Searcher Component Reference” on page 144.
Using Data Types to Narrow the Search Criteria

By default, in addition to the other search criteria, the script searches for a Content anchor according to the data type of its data holder.

For example, suppose that the search scope of a Content anchor is the following string:

```
The students' grades were 81, 56, and 95, respectively.
```

Further suppose that you define no other search criteria for the anchor. If you map the anchor to a data holder that has a type of xs:string, the anchor retrieves the entire string.

If the data holder has a type of xs:integer, the script searches for the first substring that matches the data type. Assuming that you configure the anchor with `direction = forward`, the anchor retrieves the integer 81. If `direction = backward`, the anchor retrieves 95.

Now suppose the data holder has a type of xs:integer, and the schema restricts the data holder to values less than 60. The script searches for an integer that conforms to the restriction and retrieves 56.

Data Types in Combination with Other Search Criteria

You can combine a data-type criterion with other search criteria. In the above example, suppose you configure the Content anchor to search for the following regular expression:

```
[",\*,"]
```

The expression searches for two commas, separated by any characters other than a newline. The search finds the substring

```
81, 56,
```

If the type of the data holder is xs:integer, the anchor retrieves 56.

List Data Types

A data holder can be a space-separated list. The script filters the text retrieved by the Content anchor to match the types of the list items.

Suppose that the schema defines an attribute called `grades`, which is a list of xs:integer items. In the above example, if you map the Content anchor to `grades`, the anchor returns a list of the integers in the string:

```
81 56 95
```

If the `grades` attribute belongs to an element called `Students`, the XML output is:

```
<Students grades="81 56 95" />
```

If you define the Content anchor with `direction = backward`, the list is reversed:

```
<Students grades="95 56 81" />
```

Decimal Type

If a data holder has the xs:decimal type, the script assumes that the decimal separator is a period. If your locale setting uses a comma as the decimal separator, an xs:decimal search might fail.

Type Search with Closing Marker

If a Content anchor has a closing_marker property but does not have an opening_marker, the script returns the substring closest to the closing_marker that matches the type of the data holder.
In the above example, if you define the word respectively as the closing marker, and the data holder has a type of xs:integer, the anchor retrieves 95.

Online Sample

For an online example of searching by a data type, open the project samples\Projects\Pattern\Pattern.cmw. The sample is a parser containing a single Content anchor that is mapped to an XML element. The schema uses an xs:pattern to restrict the element to certain character sequences. The anchor outputs the portion of the source document that matches the pattern.

Disabling the Data-Type Search

You can disable the data-type search by selecting the disable_XSD_type_search property of the Content anchor. If you do that, the anchor searches according to the other criteria, without regard to the type of the data holder.

If the result does not have the proper type, it cannot be stored in the data holder and the anchor fails. You can use transformers to convert the result to the proper type and prevent the failure. For more information, see "Transformers Overview" on page 156.

For example, suppose that the source document contains a date in the dd-mm-yyyy format, and you want to store the date in an xs:date data holder. You can handle this situation in the following way:

1. Define a Content anchor that retrieves the dd-mm-yyyy data, ignoring the mismatch with the xs:date type.
2. Configure the anchor with a DateFormatICU transformer that converts the result to xs:date.

Anchors that Contain Nested Anchors

An interesting question is how a parser searches for an anchor that has nested anchors, such as a Group anchor.

The script does not search for a Group, and then search for the nested anchors. Rather, it searches for the nested anchors. The extent of the Group is defined by the nested anchors that the script finds.

For example, suppose a parser has the following sequence of anchors. We assume that the anchors have default phase, marking, and optional properties.

Marker A
Group
  Marker B
  Content C
  Marker D
  Marker E

The script searches first for Marker A and Marker E. The search scope of the Group is the region between Marker A and Marker E.

Then, within the search scope of the Group, the script searches for Marker B and Marker D. The region between these Marker anchors is the search scope for Content C.

Within the latter search scope, the script searches for Content C.

Anchor Component Reference

Anchor components indicate the locations of data in the source documents.
Alternatives

The Alternatives anchor defines a set of alternative, nested anchors. You can define a criterion for selecting one alternative from the set.

The following table describes the properties of the Alternatives anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| marking | Determines whether an anchor is used as the start of the search scope for the succeeding anchor. You can choose one of the following options:  
- begin position. Place a reference point before the current anchor.  
- end position. Place a reference point after the current anchor.  
- full. Place a reference point before and after the current anchor.  
- none. Do not create a reference point.  
For more information, see “How a Parser Searches for Anchors” on page 111. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| on_fail | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| phase | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111.  
Default is main. |
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>
| selector | Determines the criterion for selecting an anchor from among the anchors nested below the **Alternatives** anchor. You can choose one of the following options:  
  - ScriptOrder. The parser tests the nested anchors in the sequence defined in the script. It accepts the first anchor that succeeds. If all the anchors fail, the **Alternatives** anchor fails.  
  - DocumentOrder. The parser tests all the nested anchors. It accepts either the first or last successful anchor, according to the locations of the anchors in the source document, as determined in the **select** property. If all the anchors fail, the **Alternatives** anchor fails.  
  - NameSwitch. The parser searches for the anchor whose **name** property is specified in the data holder defined in **option_name**. It ignores the other anchors. If the named nested anchor fails, the **Alternatives** anchor fails. |

**Example**

You are parsing a document in which a date can appear in either of the following patterns:

- 21/10/03  
- October 21, 2003

To process this content, you can define an **Alternatives** anchor that contains two **Content** anchors that store their output in different XML elements. Each XML element is constrained to accept one of the date patterns. The **Alternatives** anchor is configured with **selector = ScriptOrder**.

When the parser runs the **Alternatives** anchor, it tests the first **Content** anchor. If the date matches the pattern of the first anchor, the first **Content** anchor succeeds. If the date does not match the pattern, the first **Content** anchor fails, and the **Alternatives** anchor tests the second **Content** anchor. In this way, the parser can process both date patterns.

**How to Define an Alternatives Anchor**

Add an **Alternatives** anchor by editing the script. Nested within the **Alternatives** anchor, add the alternative anchors.

**Using Alternatives to Select a Secondary Parser**

You can use an **Alternatives** anchor to control which of several secondary parsers processes a document. The main parser can use this feature to process source documents of multiple types.

For example, suppose that the home page of a newspaper web site has links to articles. Following each link, the article is labeled **News**, **Business**, or **Sports**. You want to parse the articles, using a different parser for each type, like this:

- `<a href="PrincessWed.html">Norwegian Princess Weds</a> - News`  
- `<a href="BanksMerge.html">Local Banks to Merge</a> - Business`  
- `<a href="HomeTeamWns.html">Bears Trounce Antelopes</a> - Sports`

You can support this situation in the following way:

1. The main parser retrieves the filename of an article and stores it in a variable.
2. The main parser contains an **Alternatives** anchor that is configured with the **DocumentOrder** option.
3. The **Alternatives** anchor contains nested **Group** anchors.
4. Each **Group** anchor is configured with a **Marker** anchor and a **RunParser** action, as follows:
   - The first **Group** contains a **Marker** that searches for the string **News**. The **Group** is configured with a **RunParser** action that runs a secondary parser called **NewsParser**.
The second Group contains a Marker that searches for Business and runs BusinessParser.

The third Group contains a Marker that searches for the Sports and runs SportsParser.

The Alternatives anchor tests all three Group anchors. It accepts the Group containing the first Marker that occurs after the filename. The Group runs the appropriate parser on the file.

**Online Sample**

For an online sample of this anchor, open the project samples\Projects\Alternatives\Alternatives.cmx. The sample uses Alternatives anchors to parse different name and date formats that may exist in a source document.

**Content**

A Content anchor retrieves text from the source document. The parser searches in a defined region according to specified search criteria and stores the retrieved text in a data holder.

The following table describes the properties of the Content anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_values</td>
<td>Determines whether the Content anchor can be empty. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The data_holder is assigned an empty value.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty values are not allowed.</td>
</tr>
<tr>
<td></td>
<td>allow_empty_values must be selected in the following situations:</td>
</tr>
<tr>
<td></td>
<td>- When the anchor is configured with value = LearnByExample and there is nothing between the delimiters.</td>
</tr>
<tr>
<td></td>
<td>- When there is nothing between the opening_marker and the closing_marker.</td>
</tr>
<tr>
<td>closing_marker</td>
<td>Defines the end of a region where the parser searches for the Content anchor. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- NewlineSearch. The end of the Content anchor is the next newline character.</td>
</tr>
<tr>
<td></td>
<td>- OffsetSearch. The end of the Content anchor is the number of characters specified in offset.</td>
</tr>
<tr>
<td></td>
<td>- PatternSearch. The end of the Content anchor is the first text that matches a specified regular expression.</td>
</tr>
<tr>
<td></td>
<td>- TextSearch. The end of the Content anchor is a specified text string.</td>
</tr>
<tr>
<td>data_holder</td>
<td>Defines a data holder where the Content anchor stores the retrieved text.</td>
</tr>
<tr>
<td>direction</td>
<td>A search direction for the anchor within the search scope. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- backward. Search from the end of the search scope and finds the last instance of the anchor.</td>
</tr>
<tr>
<td></td>
<td>- forward. Search from the start of the search scope and finds the first instance of the anchor.</td>
</tr>
<tr>
<td></td>
<td>For a Marker anchor, you can modify this behavior by using the count property. For example, if direction = backward and count = 2, the script finds the second-to-last instance.</td>
</tr>
<tr>
<td></td>
<td>Default is forward. For more information, see “How a Parser Searches for Anchors” on page 111.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| disable_XSD_type_search    | Determines whether the parser searches for content that matches the data type of the data holder. You can choose one of the following options:  
  - Selected. The parser searches without regard to the data type. After transformers are applied to the content, if the result does not match the data type of the data holder, the anchor fails.  
  - Cleared. The parser searches for content that matches the data type.  
Default is cleared. For more information, see “Using Data Types to Narrow the Search Criteria” on page 116.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| disabled                   | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
The default is cleared.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| ignore_default_transformers | Determines whether the parser applies the default transformers to the content. Default is cleared.  
For more information, see “Transformers Overview” on page 156.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| marking                    | Determines whether an anchor is used as the start of the search scope for the succeeding anchor. You can choose one of the following options:  
  - begin position. Place a reference point before the current anchor.  
  - end position. Place a reference point after the current anchor.  
  - full. Place a reference point before and after the current anchor.  
  - none. Do not create a reference point.  
For more information, see “How a Parser Searches for Anchors” on page 111.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>phase</td>
<td>Determines when the script processes the component. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>• initial. The script processes the component during the initial phase.</td>
</tr>
<tr>
<td></td>
<td>• main. The script processes the component during the main phase.</td>
</tr>
<tr>
<td></td>
<td>• final. The script processes the component during the final phase.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “How a Parser Searches for Anchors” on page 111.  Default is main.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a sequence of transformers that the parser applies to the retrieved text. For more information, see “Transformers” on page 156.</td>
</tr>
<tr>
<td>validators</td>
<td>Defines a list of validators applied to the data. For more information, see “ Validators” on page 292.</td>
</tr>
<tr>
<td>value</td>
<td>Defines criteria for a search in the region defined by the opening_marker and closing_marker attributes. If opening_marker is not defined, the search is between the surrounding reference points. For more information, see “How a Parser Searches for Anchors” on page 111. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>• Empty. The Content anchor retrieves the entire search scope.</td>
</tr>
<tr>
<td></td>
<td>• AttributeSearch. The Content anchor retrieves the value from an expression of the type AttributeName=.... Use this option to retrieve attribute values from an XML or HTML source document.</td>
</tr>
<tr>
<td></td>
<td>• LearnByExample. The parser learns what text to retrieve according to the parser format and the example source. For example, if the parser has a tab-delimited format, it counts the number of tabs from the start of the search scope to the example text. It retrieves the text between the corresponding tabs in the source document.</td>
</tr>
<tr>
<td></td>
<td>• PatternSearch. The Content anchor retrieves the first text that matches a specified regular expression.</td>
</tr>
<tr>
<td></td>
<td>• TypeSearch. The Content anchor retrieves the first text that matches a specified data type.</td>
</tr>
<tr>
<td></td>
<td>Default is empty. For more information about these options, see the “Searcher Component Reference” on page 144. In addition to the searcher components, the parser uses the data type of the data_holder as a search criterion. For more information, see “Using Data Types to Narrow the Search Criteria” on page 116.</td>
</tr>
</tbody>
</table>

The opening_marker and closing_marker properties are equivalent to Marker anchors in a Group component.

- A Content anchor with the opening_marker set is like a Group component with the following sequence of anchors:
  1. Marker
  2. Content

- A Content anchor with the closing_marker set is like a Group component with the following sequence of anchors:
  1. Content
  2. Marker

- A Content anchor with the opening_marker and closing_marker set is like a Group component with the following sequence of anchors:
  1. Marker
  2. Content
  3. Marker

For more information, see the “Searcher Component Reference” on page 144.
Search Direction

The direction property has multiple effects in a Content anchor. If direction = backward:

- The script searches backward from the end of the search scope for the opening_marker and closing_marker. Opening_marker still precedes closing_marker.
- The searcher component searches backward from the end of the search scope.
- If the searcher component is LearnByExample, it counts the delimiters backward from the end of the search scope.

Online Sample

For an online sample of Content anchors, open the project samples\Projects\Content\Content.ccm. The sample illustrates several uses of the opening_marker, closing_marker, and value properties to configure Content anchors.

DelimitedSections

The DelimitedSections anchor parses data that is divided into sections by a separator. It defines a group of nested anchors. Each nested anchor parses a single section.

The following table describes the properties of the DelimitedSections anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| marking | Determines whether an anchor is used as the start of the search scope for the succeeding anchor. You can choose one of the following options:  
  - begin position. Place a reference point before the current anchor.  
  - end position. Place a reference point after the current anchor.  
  - full. Place a reference point before and after the current anchor.  
  - none. Do not create a reference point.  
  For more information, see “How a Parser Searches for Anchors” on page 111. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notifications | A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
| on_fail | The action to take if the component fails. You can choose one of the following options:  
  - Cleared. Take no action.  
  - CustomLog. Write to the user log.  
  - LogError. Write an error message to the engine log.  
  - LogInfo. Write an information message to the engine log.  
  - LogWarning. Write a warning message to the engine log.  
  - NotifyFailure. Send a notification.  
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| optional           | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
                      - Selected. Component failure does not cause the parent component to fail.  
                      - Cleared. Component failure causes the parent component to fail.  
                      Default is cleared. For more information about component failure, see “Failure Handling” on page 289.                                                                 |
| phase              | Determines when the script processes the component. You can choose one of the following options:  
                      - initial. The script processes the component during the initial phase.  
                      - main. The script processes the component during the main phase.  
                      - final. The script processes the component during the final phase.  
                      For more information, see “How a Parser Searches for Anchors” on page 111. Default is main.                                                                 |
| remark             | A user-defined comment that describes the purpose or action of the component.                                                                                                                                                                                                 |
| separator          | Defines an anchor that delimits the sections.                                                                                                                                                                                                                                 |
| separator_position | Defines the positioning of the separator relative to the sections. You can choose one of the following options:  
                      - after. There is a separator after each section, including the last section. For example,  
                        | |
|                    | 1|2|3|4 | |
| using_placeholders  | Determines when the DelimitedSections anchor looks for the separator of an optional section that is missing from the source document. You can choose one of the following options:  
                      - always. The separator of a missing section always exists. For example,  
                        | |
|                    | 1|3 | |
|                    | never. The separator of a missing section never exists. For example,  
                        | |
|                    | 1|3 | |
|                    | when necessary. The separator of a missing internal section always exists. The separator of a missing final section never exists. For example,  
                        | |
|                    | 1|3 | |

In these examples, separator_position is set to before, and sections 2 and 4 are missing.

**Example**

An employee resume form contains several sections, each of which is preceded by a line of hyphens:

```
--------------------------------------
Jane Palmer
Employee ID 123456
--------------------------------------
Professional Experience
...
--------------------------------------
Education
...
```
You can define the sectioned region as a DelimitedSections anchor, with the line of hyphens as the separator. Because the line of hyphens precedes each section, define the separator_position as before.

Within the DelimitedSections anchor, nest three Group anchors. The first Group parses the Jane Palmer section, the second Group parses the Professional Experience section, and so forth.

Optional Sections

In the above example, suppose that the second section, Professional Experience, is missing from some source documents. Its separator, the line of hyphens, is always present.

```
--
Jane Palmer
Employee ID 123456
--
Education...
```

To handle this situation, configure the DelimitedSections in the following way:

- In the second Group anchor, select the optional property. This means that if the Group fails, it does not cause the DelimitedSections to fail.
- In the DelimitedSections anchor, set using_placeholders = always. This means that the anchor looks for the separator of the optional section, even if the section itself is missing.

Now suppose that if the Professional Experience section is missing, its separator is also missing.

```
--
Jane Palmer
Employee ID 123456
--
Education...
```

In this case, configure the DelimitedSections as follows:

- In the second Group anchor, select the optional property.
- In the DelimitedSections anchor, set using_placeholders = never. This means that the anchor should not look for the separator of a missing section.

How to Define a DelimitedSections Anchor

Add a DelimitedSections anchor by editing the script in the IntelliScript editor. Under the DelimitedSections anchor, add a sequence of anchors that parse the sections.

Online Sample

For an online sample of this anchor, open the project samples\Projects\DelimitedSections\DelimitedSections.cmx. The sample illustrates a DelimitedSections anchor that parses sections separated by a | symbol. Each section is parsed by a single Content anchor.

EmbeddedParser

The EmbeddedParser anchor uses a secondary parser to parse its search scope. It can call itself recursively.
The following table describes the properties of the EmbeddedParser anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>marking</td>
<td>Determines whether an anchor is used as the start of the search scope for the succeeding anchor. You can choose one of the following options: - begin position. Place a reference point before the current anchor. - end position. Place a reference point after the current anchor. - full. Place a reference point before and after the current anchor. - none. Do not create a reference point. For more information, see “How a Parser Searches for Anchors” on page 111.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options: - Cleared. Take no action. - CustomLog. Write to the user log. - LogError. Write an error message to the engine log. - LogInfo. Write an information message to the engine log. - LogWarning. Write a warning message to the engine log. - NotifyFailure. Send a notification. Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>parser</td>
<td>Determines the name of a secondary parser that is defined in the same project.</td>
</tr>
<tr>
<td>phase</td>
<td>Determines when the script processes the component. You can choose one of the following options: - initial. The script processes the component during the initial phase. - main. The script processes the component during the main phase. - final. The script processes the component during the final phase. For more information, see “How a Parser Searches for Anchors” on page 111. Default is main.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>schema_connections</td>
<td>Defines a list of Connect subcomponents that define the relation between data holders in the output of the main parser and the secondary parser. For more information, see “Connect” on page 151.</td>
</tr>
<tr>
<td>source_transformers</td>
<td>Defines a sequence of transformers that the parser applies to the search scope before the secondary parser processes it.</td>
</tr>
</tbody>
</table>
Example

A document is tab-delimited, except for one section that is comma-delimited.

To parse the document, you can define a main parser that uses the TabDelimited format. Define another parser that uses the CommaDelimited format. Use an EmbeddedParser anchor to run the second parser within the execution of the first parser.

Online Sample

For an online sample of this anchor, open the project samples\Projects\EmbeddedParser\EmbeddedParser.cmx. The sample uses a main parser to determine the location of an address. It then runs an EmbeddedParser to parse the address.

EnclosedGroup

The EnclosedGroup anchor defines a bounded region that contains nested anchors. The boundaries are specified by opening and closing anchors. In the case of nested boundaries, such as parentheses or HTML tags, the EnclosedGroup finds the matching boundaries.

The following table describes the properties of the EnclosedGroup anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>closing</td>
<td>Defines the closing anchor of the EnclosedGroup.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>marking</td>
<td>Determines whether an anchor is used as the start of the search scope for the succeeding anchor. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- begin position. Place a reference point before the current anchor.</td>
</tr>
<tr>
<td></td>
<td>- end position. Place a reference point after the current anchor.</td>
</tr>
<tr>
<td></td>
<td>- full. Place a reference point before and after the current anchor.</td>
</tr>
<tr>
<td></td>
<td>- none. Do not create a reference point.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “How a Parser Searches for Anchors” on page 111.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>no_initial_phase</td>
<td>Determines whether the script searches for nested anchors in the main phase. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Search for nested anchors according to their individual properties.</td>
</tr>
<tr>
<td></td>
<td>- Selected. Search for nested anchors in the main phase.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notifications</td>
<td>A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307.</td>
</tr>
</tbody>
</table>

Anchor Component Reference 127
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| on_fail  | The action to take if the component fails. You can choose one of the following options:  
  - Cleared. Take no action.  
  - CustomLog. Write to the user log.  
  - LogError. Write an error message to the engine log.  
  - LogInfo. Write an information message to the engine log.  
  - LogWarning. Write a warning message to the engine log.  
  - NotifyFailure. Send a notification.  
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| opening  | Defines the opening anchor of the EnclosedGroup. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
  - Selected. Component failure does not cause the parent component to fail.  
  - Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| phase    | Determines when the script processes the component. You can choose one of the following options:  
  - initial. The script processes the component during the initial phase.  
  - main. The script processes the component during the main phase.  
  - final. The script processes the component during the final phase.  
  For more information, see “How a Parser Searches for Anchors” on page 111.  
  Default is main. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
| source   | Defines a sequence of data holders for input to the EnclosedGroup. Each data holder is identified by one of the following properties:  
  - Locator. Identifies a single-occurrence or a multiple-occurrence data holder. For multiple-occurrence data holders, each iteration accesses a new occurrence.  
  - LocatorByKey. Identifies a multiple-occurrence data holder by key.  
  - LocatorByOccurence. Identifies a multiple-occurrence data holder by sequence number.  
  Use the source property when the EnclosedGroup is called by another component. For more information, see “Source Property” on page 259. |
| target   | Defines a sequence of data holders for output from the EnclosedGroup. If a data holder does not yet exist, it is created. Each data holder is identified by one of the following properties:  
  - Locator. Identifies a single-occurrence or a multiple-occurrence data holder. For multiple-occurrence data holders, each iteration creates a new occurrence.  
  - LocatorByKey. Identifies a multiple-occurrence data holder by key.  
  - LocatorByOccurence. Identifies a multiple-occurrence data holder by sequence number.  
  Use the target property when the output of the EnclosedGroup is used by another component. For more information, see “Target Property” on page 262. |

An EnclosedGroup is similar to a Content anchor with an opening_marker and a closing_marker. However:  
- The Content anchor retrieves the entire content between the opening and closing, without further parsing.  
- The EnclosedGroup enables you to further parse the content between the opening and closing anchors.

**Example**

You can define an HTML table as an EnclosedGroup, with the `<table>` and `</table>` tags as the opening and closing. The nested anchors parse the content of the table.
Suppose the `<table>` element contains a nested `<table>` element. In other words, a table is nested within a table cell. The *EnclosedGroup* anchor matches the parent `<table>` tag with the parent `<table>` tag. It does not match the parent `<table>` tag with the nested `<table>` tag, which would be a misidentification of the table.

### How to Define an EnclosedGroup Anchor

You can define an *EnclosedGroup* anchor by editing the script in the IntelliScript editor. Add the nested anchors that parse the content.

#### ExtractRecord

The *ExtractRecord* anchor extracts a record, assigns identifiers to the record, and passes the record to the subelements of a *StructureDefinition*. *ExtractRecord* is used in the *format_definition* property of a *StructureDefinition*.

*ExtractRecord* extracts its entire search scope. For example, if you insert an *ExtractRecord* between two *Marker* anchors, it extracts the scope between the markers.

The following table describes the properties of the *ExtractRecord* anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| ids | Defines a list of identifiers attached to the record. *StructureDefinition* uses the identifiers to match the record with a subelement.  
In each list entry, enter an identifier value or browse to a data holder containing the value. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the *name* property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| phase | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111.  
Default is main. |
| remark | A user-defined comment that describes the purpose or action of the component. |

#### FindReplaceAnchor

The *FindReplaceAnchor* anchor marks the source text and specifies replacement text for transformation by the *TransformByParser* transformer.
The following table describes the properties of the **FindReplaceAnchor** anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled       | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| marking        | Determines whether an anchor is used as the start of the search scope for the succeeding anchor. You can choose one of the following options:  
- begin position. Place a reference point before the current anchor.  
- end position. Place a reference point after the current anchor.  
- full. Place a reference point before and after the current anchor.  
- none. Do not create a reference point.  
For more information, see “How a Parser Searches for Anchors” on page 111. |
| name           | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| no_initial_phase | Determines whether the script searches for nested anchors in the main phase. You can choose one of the following options:  
- Cleared. Search for nested anchors according to their individual properties.  
- Selected. Search for nested anchors in the main phase.  
Default is cleared. |
| on_fail        | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| on_partial_match | Determines the behavior when **FindReplaceAnchor** does not find all its nested, non-optional anchors. You can choose one of the following options:  
- fail. **FindReplaceAnchor** fails. Default.  
- skip. **FindReplaceAnchor** removes the area spanned by the successful nested anchors from its search scope and tries to find all the nested anchors again. It repeats this process until it finds the anchors or fails. |
| optional       | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| phase          | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111.  
Default is main. |
<p>| remark         | A user-defined comment that describes the purpose or action of the component. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace_with</td>
<td>Defines a literal replacement string or a data holder that contains the replacement string.</td>
</tr>
</tbody>
</table>
| source       | Defines a sequence of data holders for input to the FindReplaceAnchor. Each data holder is identified by one of the following properties:  
  - Locator. Identifies a single-occurrence or a multiple-occurrence data holder. For multiple-occurrence data holders, each iteration accesses a new occurrence.  
  - LocatorByKey. Identifies a multiple-occurrence data holder by key.  
  - LocatorByOccurence. Identifies a multiple-occurrence data holder by sequence number.  
  Use the source property when the FindReplaceAnchor is called by another component. For more information, see “Source Property” on page 259. |
| target       | Defines a sequence of data holders for output from the FindReplaceAnchor. If a data holder does not yet exist, it is created. Each data holder is identified by one of the following properties:  
  - Locator. Identifies a single-occurrence or a multiple-occurrence data holder. For multiple-occurrence data holders, each iteration creates a new occurrence.  
  - LocatorByKey. Identifies a multiple-occurrence data holder by key.  
  - LocatorByOccurence. Identifies a multiple-occurrence data holder by sequence number.  
  Use the target property when the output of the FindReplaceAnchor is used by another component. For more information, see “Target Property” on page 262. |

If FindReplaceAnchor does not contain nested anchors, it marks all the text within its search scope. For example, if FindReplaceAnchor is between two Marker anchors, it marks the text between them.

If FindReplaceAnchor contains a Marker anchor, it marks the Marker for replacement.

If FindReplaceAnchor contains two Marker anchors, it marks the Marker anchors and the segment between them for replacement.

The replacement text can be a static replacement string or a string retrieved dynamically from the source document.

For more information, see “TransformByParser” on page 187.

**Example**

You want to add line numbers to a text document. You can add the line numbers by the following approach:

1. Create a parser, and add a RepeatingGroup to it.
2. Within the RepeatingGroup, add a FindReplaceAnchor.
3. Within the FindReplaceAnchor, add a Marker anchor, and set its search property to NewlineSearch.
   This causes the FindReplaceAnchor to mark every newline in the document.
4. Configure the RepeatingGroup to store its current_iteration in a variable. Set the replace_with property of the FindReplaceAnchor to the variable.
5. At the global level of the script, define a TransformByParser transformer. Set its parser property to the parser.
6. Set the TransformByParser as the startup component of the transformation.
   The transformer outputs a modified version of the original file, containing line numbers.

**How to Define a FindReplaceAnchor Anchor**

You can define a FindReplaceAnchor anchor by editing the script in the IntelliScript editor. If required, add nested anchors marking a substring to be replaced.
The **Group** anchor binds a sequence of anchors and actions together.

Properties of the **Group** apply to all child components. Use a **Group** to define operations for the script to perform on a set of anchors or to control the phase of the nested anchors.

The following table describes the properties of the **Group** anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| absent           | Defines the behavior of the **Group** anchor when one of its nested, non-optional anchors or actions fails. You can choose one of the following options:  
                   - **Selected.** **Group** fails.  
                   - **Cleared.** Normal behavior.  
                   Use this feature to test for the absence of nested anchors.                                                                                                                                  |
| disabled         | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
                   - **Selected.** The script ignores the component.  
                   - **Cleared.** The script applies the component.  
                   The default is cleared.                                                                                                           |
| marking          | Determines whether an anchor is used as the start of the search scope for the succeeding anchor. You can choose one of the following options:  
                   - **begin position.** Place a reference point before the current anchor.  
                   - **end position.** Place a reference point after the current anchor.  
                   - **full.** Place a reference point before and after the current anchor.  
                   - **none.** Do not create a reference point.  
                   For more information, see “How a Parser Searches for Anchors” on page 111.                                                       |
| name             | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event.                                         |
| no_initial_phase | Determines whether the script searches for nested anchors in the main phase. You can choose one of the following options:  
                   - **Cleared.** Search for nested anchors according to their individual properties.  
                   - **Selected.** Search for nested anchors in the main phase.  
                   Default is cleared.                                                                                                                 |
| notifications    | A list of **NotificationHandler** components that handle notifications from nested components. For more information, see “Notifications” on page 307.                                                                 |
| on_fail          | The action to take if the component fails. You can choose one of the following options:  
                   - **Cleared.** Take no action.  
                   - **CustomLog.** Write to the user log.  
                   - **LogError.** Write an error message to the engine log.  
                   - **LogInfo.** Write an information message to the engine log.  
                   - **LogWarning.** Write a warning message to the engine log.  
                   - **NotifyFailure.** Send a notification.  
                   Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| on_partial_match | Determines the behavior when **Group** does not find all its nested, non-optional anchors. You can choose one of the following options:  
                   - **fail.** **Group** fails. Default.  
                   - **skip.** **Group** removes the area spanned by the successful nested anchors from its search scope and tries to find all the nested anchors again. It repeats this process until it finds the anchors or fails. |
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| phase | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111. Default is main. |
| remark | A user-defined comment that describes the purpose or action of the component. |
| search_order | Defines the direction of processing nested anchors. You can choose one of the following options:  
- top-down. The nested anchors are processed in the sequence that is defined in the script.  
- bottom-up. The nested anchors are processed in reverse order. Use this option when data from a later anchor affects the processing of an earlier anchor. |
| source | Defines a sequence of data holders for input to the Group. Each data holder is identified by one of the following properties:  
- Locator. Identifies a single-occurrence or a multiple-occurrence data holder. For multiple-occurrence data holders, each iteration accesses a new occurrence.  
- LocatorByKey. Identifies a multiple-occurrence data holder by key.  
- LocatorByOccurrence. Identifies a multiple-occurrence data holder by sequence number.  
Use the source property when the Group is called by another component. For more information, see “Source Property” on page 259. |
| target | Defines a sequence of data holders for output from the Group. If a data holder does not yet exist, it is created. Each data holder is identified by one of the following properties:  
- Locator. Identifies a single-occurrence or a multiple-occurrence data holder. For multiple-occurrence data holders, each iteration creates a new occurrence.  
- LocatorByKey. Identifies a multiple-occurrence data holder by key.  
- LocatorByOccurrence. Identifies a multiple-occurrence data holder by sequence number.  
Use the target property when the output of the Group is used by another component. For more information, see “Target Property” on page 262. |

How to Define a Group Anchor

You can define a Group anchor by editing the script in the IntelliScript editor. Add nested anchors and actions that parse the content of the Group.

Optional Group

You can use the optional property of a Group to prevent the script from attempting to retrieve text from a missing section of a document.

For example, to parse the source

```
First name: Ron
```

you might define First name: as a Marker and Ron as Content. If some source documents do not contain the first-name data, you can put the Marker and Content in a Group and make it optional. If First name: is not found, the Group immediately fails, and the parser does not search for the Content anchor.
There is a difference between making the **Group** optional and making its nested anchors optional. If you make both the **Marker** and **Content** optional, instead of the **Group**, the script ignores the **Marker** failure and searches for the **Content**. This might result in retrieving irrelevant text.

**Online Sample**

For an online sample of this anchor, open the project `samples\Projects\persistent_search\persistent_search.cmw`.

The sample illustrates a **Group** that is configured with the `on_partial_match = skip` property. The **Group** contains two **Marker** anchors:

* The first **Marker** searches for the text `A`.
* The second **Marker** searches for a string containing any number of `*` characters. It has the `adjacent` property, which means that it must be adjacent to the first **Marker**.

On the first pass, the **Group** finds an `A` character at the beginning of the source document. It does not find the second **Marker** adjacent to the `A` character, however.

The **Group** reduces its search scope by eliminating the first `A` character, and searches again for the two adjacent **Marker** anchors. It continues this procedure until it successfully finds a string `A*`, which contains the adjacent **Marker** anchors.

You can observe the behavior in the event log. The log records that the **Group** fails on the first two trials and succeeds on the third.

Try experimenting with the `on_partial_match` and `adjacent` settings. You can see the effect in the color coding of the example source.

You can also try running the sample, although the result file is empty because the parser does not contain **Content** anchors. If you set `on_partial_match = fail`, you can observe in the event log that the parser fails, because the **Group** cannot find the adjacent anchors.

**Marker**

A **Marker** anchor defines a location in a source document. It is used as a reference point, from which the script searches for the succeeding anchors.

By default, the `phase` property of a **Marker** is `initial`, which means that the script scans a document for **Marker** anchors before it searches for **Content** anchors. For more information, see "How a Parser Searches for Anchors" on page 111.
The following table describes the properties of the **Marker** anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| absent   | Determines whether the specified text or pattern is absent from the document. The **absent** property has the following options:  
  - Selected. If the specified text appears in the document, **Marker** fails.  
  - Cleared. If the specified text appears in the document, **Marker** succeeds.  
  Default is cleared. |
| adjacent | If selected, the **Marker** must be adjacent to the anchor at the beginning of its search scope. If **direction** is set to  
  - backward, it must be adjacent to the anchor at the end of its search scope. If not selected, the script can skip over text until it finds the **Marker**.  
  The **adjacent** property has the following options:  
  - Selected. The **Marker** must occur immediately after the beginning of the search scope if **direction** is set to  
    - forward or immediately before the end of the search scope if **direction** is set to **backward**.  
  - Cleared. The **Marker** can occur anywhere within the search scope.  
  Default is cleared. |
| count    | Defines the occurrence number to find. For example, to set the **Marker** at the second newline following the preceding anchor, set **search** to **NewlineSearch** and **count** to 2. |
| direction| A search direction for the anchor within the search scope. You can choose one of the following options:  
  - backward. Search from the end of the search scope and finds the last instance of the anchor.  
  - forward. Search from the start of the search scope and finds the first instance of the anchor.  
  For a **Marker** anchor, you can modify this behavior by using the **count** property. For example, if **direction** =  
    - backward and **count** = 2, the script finds the second-to-last instance.  
  Default is forward. For more information, see “How a Parser Searches for Anchors” on page 111. |
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test,  
  debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| marking  | Determines whether an anchor is used as the start of the search scope for the succeeding anchor. You can  
  choose one of the following options:  
  - begin position. Place a reference point before the current anchor.  
  - end position. Place a reference point after the current anchor.  
  - full. Place a reference point before and after the current anchor.  
  - none. Do not create a reference point.  
  For more information, see “How a Parser Searches for Anchors” on page 111. |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name**  
  property to identify the component that caused the event. |
| on_fail  | The action to take if the component fails. You can choose one of the following options:  
  - Cleared. Take no action.  
  - CustomLog. Write to the user log.  
  - LogError. Write an error message to the engine log.  
  - LogInfo. Write an information message to the engine log.  
  - LogWarning. Write a warning message to the engine log.  
  - NotifyFailure. Send a notification.  
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
## Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| phase | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase. For more information, see “How a Parser Searches for Anchors” on page 111. Default is initial. |
| remark | A user-defined comment that describes the purpose or action of the component. |
| search | Defines the search criteria for the Marker. The search criteria determine where the Marker is located within the search scope. For example, a NewlineSearch locates the Marker at a newline character. A TextSearch locates the Marker at a specified string. For more information, see “How a Parser Searches for Anchors” on page 111.  
The value of this property is one of the following searcher components:  
- NewlineSearch. Searches for a newline character.  
- TextSearch. Searches for a predefined text string or for a text string that is stored in a data holder.  
- PatternSearch. Searches for a string that matches a specified regular expression.  
- OffsetSearch. Skips a predefined number of characters following the preceding reference point, or a number of characters that is stored in a data holder. The Marker is the point following the skipped characters.  
- TypeSearch. Searches for a string that conforms to a specified data type. For more information, see the “Searcher Component Reference” on page 144. |

### How to Define a Marker Anchor

You can define a Marker by editing the script in the IntelliScript editor. For more information, see “Defining Anchors” on page 108.

### Online Sample

In the Online Samples folder, open Projects\Markers\Markers.cmw. The sample demonstrates Marker anchors that search for:

- A predefined text string
- A newline character
- An offset
- A data type
- A regular expression

If you run the parser, note that the result file is empty because the configuration does not have any Content anchors.
RepeatingGroup

The RepeatingGroup anchor parses a region that contains repetitive segments. Each segment is called an iteration, and can be delimited by a separator. The RepeatingGroup contains a sequence of nested anchors and actions that parse each iteration in the same way.

The following table describes the properties of the RepeatingGroup anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Defines a number or data holder that contains the number of iterations to run. If blank, the iterations continue until the search scope is exhausted. If count is 0, the RepeatingGroup does not search for iterations. In this case, the RepeatingGroup succeeds but does not produce any output.</td>
</tr>
<tr>
<td>current_iteration</td>
<td>Defines a data holder where the RepeatingGroup outputs the number of the current iteration.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>iteration_order</td>
<td>Defines the order in which the iterations are processed. You can choose one of the following options: - top-down. The iterations are processed in the sequence that is defined in the script. - bottom-up. The iterations are processed in reverse order. Use this option if data from a later iteration affects how you process an earlier iteration.</td>
</tr>
<tr>
<td>marking</td>
<td>Determines whether an anchor is used as the start of the search scope for the succeeding anchor. You can choose one of the following options: - begin position. Place a reference point before the current anchor. - end position. Place a reference point after the current anchor. - full. Place a reference point before and after the current anchor. - none. Do not create a reference point. For more information, see “How a Parser Searches for Anchors” on page 111.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>no_initial_phase</td>
<td>Determines whether the script searches for nested anchors in the main phase. You can choose one of the following options: - Cleared. Search for nested anchors according to their individual properties. - Selected. Search for nested anchors in the main phase. Default is cleared.</td>
</tr>
<tr>
<td>notifications</td>
<td>A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307.</td>
</tr>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options: - Cleared. Take no action. - CustomLog. Write to the user log. - LogError. Write an error message to the engine log. - LogInfo. Write an information message to the engine log. - LogWarning. Write a warning message to the engine log. - NotifyFailure. Send a notification. Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>on_iteration_fail</td>
<td>Defines the action when a single iteration fails. You can choose one of the following options: - Cleared. No action. - CustomLog. Writes to the user log. - LogError. Writes an error message to the Engine log. - LogInfo. Writes an information message to the Engine log. - LogWarning. Writes a warning message to the Engine log. - NotifyFailure. Triggers a notification. Use the <strong>on_fail</strong> property to write an entry if the entire <strong>RepeatingGroup</strong> fails. For more information, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>on_partial_match</td>
<td>Defines the behavior when some, but not all, of the required anchors nested under the <strong>RepeatingGroup</strong> appear in the input. The <strong>on_partial_match</strong> property has the following options: - fail. The iteration fails. - skip. <strong>RepeatingGroup</strong> removes the area spanned by the successful nested anchors from its search scope and tries to find all the nested anchors again. The removal-retry procedure is repeated until the iteration succeeds or until there is no longer a partial match. If there is no partial match, the iteration fails.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>phase</td>
<td>Determines when the script processes the component. You can choose one of the following options: - initial. The script processes the component during the initial phase. - main. The script processes the component during the main phase. - final. The script processes the component during the final phase. For more information, see “How a Parser Searches for Anchors” on page 111. Default is main.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>search_order</td>
<td>Defines the order of processing the nested anchors within each iteration. You can choose one of the following options: - top-down. The nested anchors are processed in the sequence that is defined in the script. - bottom-up. The nested anchors are processed in reverse order. Select this option if data from a later anchor affects how you process an earlier anchor.</td>
</tr>
<tr>
<td>separator</td>
<td>Defines an anchor that delimits the sections.</td>
</tr>
<tr>
<td></td>
<td>If you leave the <strong>separator</strong> property empty, the <strong>RepeatingGroup</strong> does not look for a delimiter between the iterations. Instead, it assumes that an iteration is finished when it has found all the nested anchors. It then starts to parse the next iteration from the top of the nested anchor sequence. You can build a complex separator by inserting a <strong>Group</strong> in the <strong>separator</strong> property instead of a <strong>Marker</strong>.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>separator_position</td>
<td>Defines the positioning of the separator relative to the sections. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- after. There is a separator after each section, including the last section. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>- around. There are separators before and after each section, including the first and last sections. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>- before. There is a separator before each section, including the first section. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>- between. There is a separator between the successive sections, but not before the first section and not after the last section. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>skip_failed_iterations</td>
<td>Determines whether failed iterations are skipped. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. RepeatingGroup skips over a failed iteration and proceeds with the next iteration. If an iteration succeeds, the RepeatingGroup succeeds.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. RepeatingGroup fails if any iteration fails.</td>
</tr>
<tr>
<td></td>
<td>The skip_failed_iterations property has an effect only if separator is defined. Default is selected.</td>
</tr>
<tr>
<td>source</td>
<td>Defines a sequence of data holders for input to the RepeatingGroup. Each data holder is identified by one of the following properties:</td>
</tr>
<tr>
<td></td>
<td>- Locator. Identifies a single-occurrence or a multiple-occurrence data holder. For multiple-occurrence data holders, each iteration accesses a new occurrence.</td>
</tr>
<tr>
<td></td>
<td>- LocatorByKey. Identifies a multiple-occurrence data holder by key.</td>
</tr>
<tr>
<td></td>
<td>- LocatorByOccurence. Identifies a multiple-occurrence data holder by sequence number.</td>
</tr>
<tr>
<td></td>
<td>Use the source property when the RepeatingGroup is called by another component. For more information, see “Source Property” on page 259.</td>
</tr>
<tr>
<td>target</td>
<td>Defines a sequence of data holders for output from the RepeatingGroup. If a data holder does not yet exist, it is created. Each data holder is identified by one of the following properties:</td>
</tr>
<tr>
<td></td>
<td>- Locator. Identifies a single-occurrence or a multiple-occurrence data holder. For multiple-occurrence data holders, each iteration creates a new occurrence.</td>
</tr>
<tr>
<td></td>
<td>- LocatorByKey. Identifies a multiple-occurrence data holder by key.</td>
</tr>
<tr>
<td></td>
<td>- LocatorByOccurence. Identifies a multiple-occurrence data holder by sequence number.</td>
</tr>
<tr>
<td></td>
<td>Use the target property when the output of the RepeatingGroup is used by another component. For more information, see “Target Property” on page 262.</td>
</tr>
</tbody>
</table>

**Note:** To parse a region of sections that require differing treatment, use a DelimitedSections anchor.

**How to Define a RepeatingGroup Anchor**

You can define a RepeatingGroup by editing the script in the IntelliScript editor. Add the nested anchors and actions that parse each iteration of the RepeatingGroup.

**Search for Iterations**

By default, a RepeatingGroup searches for iterations from the beginning to the end of its search scope. For more information, see “How a Parser Searches for Anchors” on page 111.

 Optionally, you can set the iteration_order property for a reverse search.

In each iteration:

- If the RepeatingGroup is configured with a separator, it searches for the next separator. Then, it searches for the anchors lying between a pair of separators.
End of a RepeatingGroup

You can signal the end of a RepeatingGroup in ways such as the following:

- If the RepeatingGroup does not have a separator, it ends when the parser cannot find any more iterations.

Success or Failure of a RepeatingGroup

If a RepeatingGroup cannot find the non-optional anchors in an iteration, the iteration fails.

When an iteration fails, the RepeatingGroup can either end, fail, or skip the failed iteration. The behavior is as follows:

- If the RepeatingGroup does not have a separator, the RepeatingGroup ends. Provided that there was at least one successful iteration prior to the failed iteration, the RepeatingGroup succeeds.
- If the RepeatingGroup has a separator, and the skip_failed_iterations property is not selected, the RepeatingGroup fails.
- If the RepeatingGroup has a separator, and the skip_failed_iterations property is selected, the script skips over the failed iteration and proceeds with the next iteration. Provided that at least one iteration succeeds, the RepeatingGroup succeeds.

Event Log of a Repeating Group

The event log records events for every iteration of a RepeatingGroup.

If the skip_failed_iterations property is selected, the RepeatingGroup might generate an optional failure event following the successful iterations. A failure event might be nested within the optional failure. These events occur because the RepeatingGroup cannot find additional iterations to parse. The events are normal and not a cause for concern.

Online Samples

For an online example of this anchor, open the project samples\Projects\Dynamic_And_RepeatingGroup\Dynamic_And_RepeatingGroup.cmw. The sample uses a RepeatingGroup to iterate over the lines of a document.

Some lines of the source document contain a parenthesized footnote reference, such as "(1)". The RepeatingGroup contains a Group, whose purpose is to parse the footnote and insert its content in the output.

The Group contains a Content anchor that retrieves the footnote reference and stores it in a variable. The Group then activates a RunParser action that activates a secondary parser. The secondary parser finds the footnote referenced by the variable, parses it, and inserts the result in the output.

StructureDefinition

The StructureDefinition anchor processes well-structured input, such as text messages conforming to industry-standard messaging protocols. The output of StructureDefinition is an XML representation of the data.
The input data must have records that are delimited and organized in predefined ways. For example, a record of type A must be followed by a record of type B, or by one to three records of type C. Each record contains an organized set of fields.

The following table describes the properties of the StructureDefinition anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>format_definition</td>
<td>Defines a list of anchors and actions that identify and extract the records. The list must contain an ExtractRecord anchor.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>notifications</td>
<td>A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307.</td>
</tr>
<tr>
<td>phase</td>
<td>Determines when the script processes the component. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- initial. The script processes the component during the initial phase.</td>
</tr>
<tr>
<td></td>
<td>- main. The script processes the component during the main phase.</td>
</tr>
<tr>
<td></td>
<td>- final. The script processes the component during the final phase.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “How a Parser Searches for Anchors” on page 111.</td>
</tr>
<tr>
<td></td>
<td>Default is main.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>target</td>
<td>Defines a data holder where the component stores its output.</td>
</tr>
</tbody>
</table>

A StructureDefinition has the following parts:

- format_definition property. Extracts the records and identifies their types.
- A hierarchy of child components. Each child component parses records of a particular type.

The format_definition can contain a RepeatingGroup that finds the records. Within the RepeatingGroup, one or more Content anchors retrieve the record-type identifiers. The RepeatingGroup contains an ExtractRecord anchor that passes the record to the subelements list.

The subelements hierarchy reflects the required organization of the records. You can configure sequences, choices, and loops of records, as well as mandatory and optional records.

The subelements receive the records from ExtractRecord. The system matches each record to a subelement according to the following criteria:

- The $id and $qualifier identifiers of the record must match the values specified in the subelement.
- The record location in the input must match the subelement location in the hierarchy.

The matching subelement parses the record.

If there is no matching subelement, the StructureDefinition triggers a notification. You can insert NotificationHandler components that process the notification. In this way, StructureDefinition helps to locate and diagnose input errors. In most cases, StructureDefinition can continue to parse the remainder of the input.
The second subelement defines a sequence of nested subelements. The first nested subelement matches a record having the identifiers

You can parse this input by configuring a StructureDefinition anchor.

The format_definition property contains a RepeatingGroup that finds the records. The RepeatingGroup performs the following operations:

1. It finds the record content, up to the newline delimiter.
2. It extracts the record-type identifier, such as ST or NM1, and stores it in the $id variable.
3. If the record type is NM1, it extracts the subtype (12 or 13), and stores it in the $qualifier variable.
4. It runs an ExtractRecord anchor that passes the record to the subelements. ExtractRecord attaches the $id and $qualifier identifiers to the record.

The element is configured to match any record that has the identifier ST.

The format_definition encounters the first input record and passes it to the subelements. The first record matches the first subelement because it has the ST identifier. The first subelement contains Content Marker Content anchors, which parse the record.

The second subelement defines a sequence of nested subelements. The first nested subelement matches a record having the identifiers NM1 and 12. The second nested subelement matches a record having an identifier of N1.

The second and third input records are NM1*12 and N1. These match the sequence of subelements. Each nested subelement parses the corresponding record.

Suppose that the second and third records were NM1*12 and N2. These do not match the subelements hierarchy, so they would not be parsed.

The next records are NM1*13 and N2. They match the third subelement, named Loop2000.

The last record is SE, matching the last subelement.
All the input records match the subelements hierarchy, so the **StructureDefinition** successfully parses the complete input.

### Subelement Components

Within the subelements hierarchy, you can insert the following components:

<table>
<thead>
<tr>
<th>Subelement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordStructureLocal</td>
<td>Matches and parses a single record.</td>
</tr>
<tr>
<td>SequenceStructureLocal</td>
<td>Defines a sequence of nested subelements. The records must occur in the same sequence as the nested subelements.</td>
</tr>
<tr>
<td>ChoiceStructureLocal</td>
<td>Defines a choice of nested subelements. A record must match one of the nested subelements.</td>
</tr>
<tr>
<td>AllStructureLocal</td>
<td>Defines a set of nested subelements, without a specified sequence. The records can match the nested subelements in any order.</td>
</tr>
</tbody>
</table>

The names end with `Local` because you can configure them in nested, non-global locations of the script. There is a corresponding set of components called `RecordStructure`, `SequenceStructure`, and so forth, without the `local` suffix. You can configure these at the global level of the script and reference them wherever required. To reference the global components, insert an `EmbeddedStructure` subelement.

The top-level subelements list of a `StructureDefinition` is equivalent to `SequenceStructureLocal`. The records must occur in the same sequence as the top-level subelements.

By default, each subelement must occur exactly once. To alter the default, set the `minOccurs` and `maxOccurs` properties of the subelement. For example, if a subelement can be missing or occur up to 3 times, set `minOccurs = 0` and `maxOccurs = 3`. To permit unlimited occurrences, set `maxOccurs = -1`.

For more information about the subelement components, see the “Anchor Subcomponent Reference” on page 148.

### Notifications

If a record or a set of records does not match the subelements hierarchy, **StructureDefinition** triggers a notification.

The following table describes the types of notifications:

<table>
<thead>
<tr>
<th>Notification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MandatoryStructureMissing</td>
<td>A mandatory record does not appear in the input.</td>
</tr>
<tr>
<td>MismatchIDs</td>
<td>The record and subelement IDs partially match. For example, there are two record identifiers, and only one of them matches.</td>
</tr>
<tr>
<td>StructureBelowMinOccurs</td>
<td>There are fewer matching records of the subelement than defined in <code>minOccurs</code>.</td>
</tr>
<tr>
<td>StructureExceedsMaxOccurs</td>
<td>There are more matching records of the subelement than defined in <code>maxOccurs</code>.</td>
</tr>
<tr>
<td>StructureOutOfSequence</td>
<td>The records match the subelements but not in the required sequence. For example, the subelements define a sequence <code>ABC</code>, but the input contains <code>ACB</code>.</td>
</tr>
</tbody>
</table>
### Notification

<table>
<thead>
<tr>
<th>Notification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnexpectedRecord</td>
<td>The records match the subelements, but not in the required hierarchy. For example, the subelement define a sequence ABC, and D is defined in another location. The input contains ABD.</td>
</tr>
<tr>
<td>UnrecognizedRecord</td>
<td>No subelement matches any of the record identifiers.</td>
</tr>
<tr>
<td>XsdValidationError</td>
<td>The input does not match the requirements of the schema.</td>
</tr>
</tbody>
</table>

Configure the NotificationHandler components in the notifications property of the StructureDefinition or a subelement. You can also configure handlers in the notifications property of a higher-level component such as a Parser or Group that contains the StructureDefinition. If a handler exists in the subelement where a mismatch occurs, it processes the notification. If no handler exists, the notification bubbles up the IntelliScript hierarchy until a handler processes it. If there is no handler for a notification, the notification is ignored and the StructureDefinition continues processing the input.

**Keeping Track of Progress**

As the format_definition extracts records, it updates the VarStructureDetails system variable. You can use the variable in notifications. For example, to report the record identifier, a notification handler can insert VarStructureDetails/RecordId in its output.

For more information, see “System Variables” on page 99.

### Searcher Component Reference

Searcher components are used for the following purposes:

- To define the location of anchors. For more information, see “Anchor Component Reference” on page 117.
- To define delimiter characters or strings. For more information, see “Format Component Reference” on page 80.
- To define the find what string of a Replace transformer. For more information, see “Transformer Component Reference” on page 158.

**AttributeSearch**

The AttributeSearch searcher component searches a source document for the value of a specified attribute. The component retrieves the value from an expression in one of the following formats:

- `AttributeName = value`
- `AttributeName = "value"`

where `AttributeName` is the name of the attribute, the quotes can be single or double, and the spaces are optional.

AttributeSearch is one of the settings of the value property of the Content anchor. For more information, see “Content” on page 120.
The following table describes the properties of the **AttributeSearch** searcher component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>att</td>
<td>Defines the name of the attribute.</td>
</tr>
</tbody>
</table>
| match_case | Determines whether the name of the attribute is case sensitive. The **match_case** property has the following options:  
  - Selected. The name of the attribute is case sensitive.  
  - Cleared. The name of the attribute is not case sensitive. |

**Example**

An HTML document contains the element:

```html
<img src='MyPicture.gif'>
```

You can use **AttributeSearch** to retrieve the value of the **src** attribute. It returns the text **MyPicture.gif**.

**Valid Attribute Syntax**

**AttributeSearch** reads name-value pairs that contain an equals sign. The equals sign can be surrounded by spaces. The value can be surrounded by double quotes, single quotes, or no quotes.

For example, suppose that **AttributeSearch** is configured to search for an attribute called **time**. All the following examples have valid syntax and return the same value, **12:55:33**.

```plaintext
  time = "12:55:33"
  time="12:55:33"
  time = '12:55:33'
  time='12:55:33'
  time = 12:55:33
  time=12:55:33
```

**Online Sample**

For an online sample of this component, open the project **samples\Projects\Content\Content.cmw**. The sample illustrates the use of an **AttributeSearch** to parse a text document that has a **variable = value** structure.

**LearnByExample**

The **LearnByExample** searcher component learns how to search for text by examining the text location in the example source document. It uses the parser format to interpret the source document.

For example, if the parser has a tab-delimited format, **LearnByExample** counts the number of tabs from the search start to the example text. It searches for text in the source document that lies at the same number of tabs from the start of the search scope.

**LearnByExample** is one of the settings of the **value** property of the **Content** anchor. For more information, see "**Content**" on page 120.

If the **direction** attribute of the **Content** anchor is set to **backward**, the component counts the delimiters from the end of the search scope.
The following table describes the properties of the **LearnByExample** searcher component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>example</td>
<td>Defines the text in the example source document at the anchor location.</td>
</tr>
</tbody>
</table>

**NewlineSearch**

The **NewlineSearch** searcher component searches for a newline or linefeed character (0x0A), a carriage return character (0x0D), or both.

The **Marker** anchor can use **NewlineSearch** to find newline markers. A **Delimiter** component can use **NewlineSearch** to find newline delimiters.

**OffsetSearch**

The **OffsetSearch** searcher component defines the number of characters between a reference point and an anchor. For example, it can define the number of characters between the end of a **Marker** and the start of a **Content** anchor.

The following table describes the properties of the **OffsetSearch** searcher component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_smaller_offset</td>
<td>Determines whether an offset that extends beyond the search scope is valid. Select this property to permit a truncated field size at the end of a document. You can choose one of the following options: - Selected. <strong>OffsetSearch</strong> succeeds when an offset extends beyond the search scope. - Cleared. <strong>OffsetSearch</strong> fails when an offset extends beyond the search scope.</td>
</tr>
<tr>
<td>offset</td>
<td>Defines the number of characters between the reference point and the anchor. In some locations where <strong>OffsetSearch</strong> is used, such as in a <strong>Marker</strong> anchor, the IntelliScript editor displays a browse button next to the <strong>offset</strong> property. You can enter a value or browse to a data holder containing the value.</td>
</tr>
</tbody>
</table>

**PatternSearch**

The **PatternSearch** searcher component searches for a string that matches a regular expression.

Anchors can use **PatternSearch** to find markers or content. The **Delimiter** component can use **PatternSearch** to find delimiters. The **Replace** transformer can use **PatternSearch** to find the text to be replaced.

The following table describes the properties of the **PatternSearch** searcher component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape_sequence</td>
<td>Defines a prefix that causes the search component to ignore an instance of the pattern in the source document.</td>
</tr>
<tr>
<td>pattern</td>
<td>Defines the regular expression.</td>
</tr>
</tbody>
</table>

For more information about the syntax of regular expressions, see “Regular Expression Syntax” on page 180.
Example

Suppose you want to define the string `%%`, containing one or more `%` symbols, as a delimiter. Within the **Delimiter** component, you can use **PatternSearch** with the following regular expression:

```
%%
```

In another example, suppose you want to define a comma and a semicolon as alternative delimiters, at the same level of the delimiter hierarchy. You can use the following regular expression:

```
[,];
```

SegmentSearch

The **SegmentSearch** searcher component searches for opening and closing markers in a text string. It returns the segment from the opening marker to the closing marker, including the markers themselves. **SegmentSearch** is one of the options for the `find_what` attribute of the **Replace** transformer.

The following table describes the properties of the **SegmentSearch** searcher component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opening</td>
<td>Defines the search criterion for the opening marker. The options are the following searcher components:</td>
</tr>
<tr>
<td></td>
<td>- NewlineSearch</td>
</tr>
<tr>
<td></td>
<td>- OffsetSearch</td>
</tr>
<tr>
<td></td>
<td>- PatternSearch</td>
</tr>
<tr>
<td></td>
<td>- TextSearch</td>
</tr>
<tr>
<td>closing</td>
<td>Defines the search criterion for the closing marker. The options are the following searcher components:</td>
</tr>
<tr>
<td></td>
<td>- NewlineSearch</td>
</tr>
<tr>
<td></td>
<td>- OffsetSearch</td>
</tr>
<tr>
<td></td>
<td>- PatternSearch</td>
</tr>
<tr>
<td></td>
<td>- TextSearch</td>
</tr>
</tbody>
</table>

TextSearch

The **TextSearch** searcher component searches for an explicit string.

Anchors can use **TextSearch** to find markers. The **Delimiter** component can use **TextSearch** to find delimiters. The **Replace** transformer can use **TextSearch** to find text that is to be replaced.

The following table describes the properties of the **TextSearch** searcher component:

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape_sequence</td>
<td>Defines a prefix that causes the search to ignore an instance of the string in the source document. In locations where dynamic search is supported, you can browse to a data holder that contains the escape sequence.</td>
</tr>
<tr>
<td>match_case</td>
<td>Determines whether the defined <code>text</code> must match exactly, with the same uppercase and lowercase letters. Default is cleared.</td>
</tr>
<tr>
<td>text</td>
<td>Defines the string to find. In locations where dynamic search is supported, you can browse to a data holder that contains the string.</td>
</tr>
</tbody>
</table>
**Example**

To define the string percent-percent-tab as a delimiter, create a **Delimiter** component and set its **search** property to **TextSearch**. In the **text** property, type:

```%
%%
```

Then press **CTRL+A** and type **009** (the ASCII code of a tab character).

**Specifying a Search String Dynamically**

In some locations where **TextSearch** is used, such as in a **Delimiter** component or a **Marker** anchor, a browse button appears to the right of the text box. Browse to a data holder that contains the search text.

To find repeated instances of the first word in a document, you can define a **Content** anchor that retrieves the first word and stores it in a variable. You can then define **Marker** anchors that use **TextSearch** to find other instances of the word that you stored in the variable.

**Online Sample**

For an online sample of this component, open the project `samples\Projects\Dynamic_And_RepeatingGroup\Dynamic_And_RepeatingGroup.cmw`.

In the **GetRemarkParser** component of this sample, a **Marker** anchor uses a dynamically defined **TextSearch** to find a footnote at the end of the source document. For more information about this sample, see “RepeatingGroup” on page 137.

**TypeSearch**

The **TypeSearch** searcher component searches for an anchor of a specified data type.

**TypeSearch** is one of the settings of the **value** property of the **Content** anchor. For more information, see “Content” on page 120.

The following table describes the properties of the **TypeSearch** searcher component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val_type</td>
<td>Determines the data type of the anchor to search for.</td>
</tr>
</tbody>
</table>

**Anchor Subcomponent Reference**

Anchor subcomponents are assigned as the values of certain anchor properties.

**AllStructure**

The **AllStructure** component defines a set of nested sub-elements without regard to sequence. A set of records matches **AllStructure** if it matches all sub-elements in any sequence. For more information, see “StructureDefinition” on page 140.

The **AllStructure** component appears at the global level of the script and has the same function as **AllStructureLocal**. You can reference it in the **ref** attribute of an **EmbeddedStructure**.
The following table describes the properties of the AllStructure component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>Defines an action that runs on the list of sub-components.</td>
</tr>
</tbody>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notifications | A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
| target   | Defines a data holder where the component stores its output. |

AllStructureLocal

The AllStructureLocal component defines a set of nested sub-elements without regard to sequence. A set of records matches AllStructureLocal if it matches all sub-elements in any sequence. For more information, see “StructureDefinition” on page 140.

AllStructureLocal is a sub-element of the StructureDefinition anchor and has the same function as AllStructure. At the global level of the script, use AllStructure.

The following table describes the properties of the AllStructure component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>Defines an action that runs on the list of sub-components.</td>
</tr>
</tbody>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| minOccurs | Defines the minimum number of matching records. Default is 1. |
| maxOccurs | Defines the maximum number of matching records. Default is 1. Use -1 for an unlimited number. |
| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notifications | A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
The **ChoiceStructure** component defines a set of nested sub-elements. A record matches **ChoiceStructure** if it matches any nested sub-element. For more information, see “StructureDefinition” on page 140.

The **ChoiceStructure** component appears at the global level of the script and has the same function as **ChoiceStructureLocal**. You can reference it in the `ref` attribute of an **EmbeddedStructure**.

The following table describes the properties of the **ChoiceStructure** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>Defines an action that runs on the list of sub-components.</td>
</tr>
</tbody>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the `name` property to identify the component that caused the event. |
| notifications | A list of **NotificationHandler** components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
| target   | Defines a data holder where the component stores its output. |

**ChoiceStructureLocal**

The **ChoiceStructureLocal** component defines a set of nested sub-elements. A record matches **ChoiceStructureLocal** if it matches any nested sub-element. For more information, see “StructureDefinition” on page 140.

**ChoiceStructureLocal** is a sub-element of the **StructureDefinition** anchor and has the same function as **ChoiceStructure**. At the global level of the script, use **ChoiceStructure**.
The following table describes the properties of the `ChoiceStructureLocal` component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>Defines an action that runs on the list of sub-components.</td>
</tr>
</tbody>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| minOccurs | Defines the minimum number of matching records. Default is 1. |
| maxOccurs | Defines the maximum number of matching records. Default is 1. Use -1 for an unlimited number. |
| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the `name` property to identify the component that caused the event. |
| notifications | A list of `NotificationHandler` components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
| sub_elements | Defines a list of nested sub-elements. |
| target   | Defines a data holder where the component stores its output. |

**Connect**

The `Connect` component specifies a link between data holders in two components. The two data holders must have the same data type.

The following table describes the properties of the `Connect` component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_holder</td>
<td>Defines a data holder that is referenced in the main parser, serializer, or mapper. <strong>Note:</strong> If the data holder is a variable, the transformation assigns it an empty default value. If the variable has a data type that does not accept an empty value, such as <code>xs:boolean</code>, ensure that the variable has a value before you run the embedded transformation.</td>
</tr>
<tr>
<td>embedded_data_holder</td>
<td>Defines a data holder that is referenced in the secondary parser, serializer, or mapper.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the <code>name</code> property to identify the component that caused the event.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

The `schema_connections` property of the following components can have one or more instances of the `Connect` component:

- EmbeddedParser. Specifies where a secondary parser stores its result in the output of the main parser.
- EmbeddedSerializer. Specifies a link between the input data holders of a secondary serializer and the input data holders of the main serializer.
- EmbeddedMapper. Specifies a link between the input and output data holders.
- EmbeddedStructure. Specifies a link between the targets of global and local StructureDefinition sub-elements.

Example

A secondary parser outputs an XML element called ID. You want the main parser to store this result in a variable called VarID. You can connect ID to VarID.

For an additional example, see "EmbeddedSerializer" on page 237.

EmbeddedStructure

The EmbeddedStructure component activates components defined at the global level of the script. For more information, see "StructureDefinition" on page 140.

EmbeddedStructure is a sub-element of the StructureDefinition anchor.

The following table describes the properties of the EmbeddedStructure component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled       | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:
|                | - Selected. The script ignores the component.
|                | - Cleared. The script applies the component.
|                | The default is cleared. |
| minOccurs       | Defines the minimum number of matching records. |
| maxOccurs      | Defines the maximum number of matching records. Use -1 for an unlimited number. |
| name           | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notifications  | A list of NotificationHandler components that handle notifications from nested components. For more information, see "Notifications" on page 307. |
| ref            | Defines the name of the globally configured component. |
| remark         | A user-defined comment that describes the purpose or action of the component. |
| schema_connection | Connects the target of the references subelement with the target of the EmbeddedStructure. For more information, see "Connect" on page 151. |
| target         | Defines a data holder where the component stores its output. |

RecordStructure

The RecordStructure component defines a set of components. A set of records matches RecordStructure if it has the same identifiers. For more information, see "StructureDefinition" on page 140.
The RecordStructure component appears at the global level of the script and has the same function as RecordStructureLocal. You can reference it in the ref attribute of an EmbeddedStructure.

The following table describes the properties of the RecordStructure component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>ids</td>
<td>Defines one or more strings.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>notifications</td>
<td>A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>target</td>
<td>Defines a data holder where the component stores its output.</td>
</tr>
</tbody>
</table>

RecordStructureLocal

The RecordStructureLocal component defines a set of components. A set of records matches RecordStructureLocal if it has the same identifiers. For more information, see “StructureDefinition” on page 140.

RecordStructureLocal is a sub-element of the StructureDefinition anchor and has the same function as RecordStructure. At the global level of the script, use RecordStructure.

The following table describes the properties of the RecordStructureLocal component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>ids</td>
<td>Defines one or more strings.</td>
</tr>
<tr>
<td>minOccurs</td>
<td>Defines the minimum number of matching records. Default is 1.</td>
</tr>
<tr>
<td>maxOccurs</td>
<td>Defines the maximum number of matching records. Default is 1. Use -1 for an unlimited number.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>notifications</td>
<td>A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307.</td>
</tr>
</tbody>
</table>
## SequenceStructure

The **SequenceStructure** component defines a sequence of nested sub-elements. A set of records matches **SequenceStructure** if it matches all nested sub-elements in sequence. For more information, see “StructureDefinition” on page 140.

The **SequenceStructure** component appears at the global level of the script and has the same function as **SequenceStructureLocal**. You can reference it in the **ref** attribute of an **EmbeddedStructure**.

The following table describes the properties of the **SequenceStructure** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>target</td>
<td>Defines a data holder where the component stores its output.</td>
</tr>
</tbody>
</table>

## SequenceStructureLocal

The **SequenceStructureLocal** component defines a sequence of nested sub-elements. A set of records matches **SequenceStructureLocal** if it matches all nested sub-elements in sequence. For more information, see “StructureDefinition” on page 140.

**SequenceStructureLocal** is a sub-element of the **StructureDefinition** anchor and has the same function as **SequenceStructure**. At the global level of the script, use **SequenceStructure**.
The following table describes the properties of the `SequenceStructureLocal` component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>Defines an action that runs on the list of sub-components.</td>
</tr>
</tbody>
</table>
| disabled       | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| minOccurs       | Defines the minimum number of matching records. Default is 1.                                                                                 |
| maxOccurs      | Defines the maximum number of matching records. Default is 1. Use -1 for an unlimited number.                                                |
| name           | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notifications  | A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
| remark         | A user-defined comment that describes the purpose or action of the component.                                                                |
| sub_elements   | Defines a list of nested sub-elements.                                                                                                       |
| target         | Defines a data holder where the component stores its output.                                                                               |
This chapter includes the following topics:

- Transformers Overview, 156
- Defining Transformers, 156
- Standard Transformer Properties, 158
- Transformer Component Reference, 158

Transformers Overview

Transformers modify the output of other components.

You can use transformers within components such as anchors, serialization anchors, and actions. For example, if you use a transformer within a Content anchor, it modifies the data that the anchor extracts from the source document.

You can use transformers as document processors. You can also define a transformer at the global level of a script and set it as the startup component.

Defining Transformers

You can define transformers in the following locations of the script:

- In the transformers property of an anchor or a serialization anchor
- In the default_transformers property of a format or of a serializer
- In the ProcessByTransformers document processor
- In the transformers property of certain actions
- At the global level, as a standalone, runnable component that modifies a source document.

Using Transformers in Anchors

You can use transformers in an anchor that creates XML output, such as Content. In the script, nest the transformer components within the transformers property of the anchor.

The input of a transformer is the raw output of the anchor, before the anchor inserts the output in a data holder.
For example, suppose you are parsing the following source document:

```
First name: Ron
Last name: Lehrer
```

You want to create XML output in ALL CAPS, like this:

```
<Person>
  <FirstName>RON</FirstName>
  <LastName>LEHRER</LastName>
</Person>
```

To do this, you can configure the Content anchors, which retrieve the strings Ron and Lehrer, with the ChangeCase transformer.

### Sequences of Transformers

You can configure an anchor with a sequence of transformers. Each transformer modifies the output of the preceding transformer.

In the Ron Lehrer example, suppose you want the following output:

```
<Person>
  <FirstName>- RON -</FirstName>
  <LastName>- LEHRER -</LastName>
</Person>
```

To do this, you might configure the Content anchors with the ChangeCase and AddString transformers. The transformers change the case and add the hyphens, in sequence.

### Default Transformers

Very often, you want the same transformers to run on all the Content anchors in a parser. You can configure the format component of the parser with default transformers. This saves you the trouble of adding the same transformers to every anchor in the parser.

To do this, nest the transformers in the `default_transformers` property of the format. For more information, see “Format Component Reference” on page 80.

Many of the predefined format components include default transformers. For example, the HtmlFormat component has default transformers that remove HTML tags from the output and convert HTML entities to plain text. You can change the default transformers by editing the `default_transformers` property.

If an anchor has its own transformers, they run after the default transformers.

You can cancel the default transformers for particular anchors. To do this, set the `ignore_default_transformers` property of the anchor.

### Using Transformers as Document Processors

You can run a transformer or a sequence of transformers as a document processor.

For example, you might run the RemoveTags transformer as a processor on an HTML document. The transformer removes the HTML tags before a parser starts to search for anchors in the document.

To do this, configure the parser format component with the ProcessByTransformers document processor, and nest the transformers within the component.
Using Transformers in Serialization Anchors

You can use transformers in serialization anchors that write to the output document, such as ContentSerializer. The transformers modify the data before the serializer writes it to the document.

For example, a ContentSerializer might write the content of a data holder called DoctorName to an output document. You might configure the ContentSerializer with an AddString transformer that adds the prefix "Dr. " to the content. Suppose the XML input has the following form:

```xml
<DoctorName>Albert Schweitzer</DoctorName>
```

The transformer modifies the content, resulting in the following output:

```
Dr. Albert Schweitzer
```

You can add transformers to the default_transformers property of a serializer. The transformers that you add here run in all the ContentSerializer serialization anchors before they write to the output document.

Using Transformers in Actions

Certain actions, such as SetValue and Map, apply transformers to their output. For more information, see “Actions Overview” on page 192.

Standard Transformer Properties

The following table describes standard properties of transformers:

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |

Transformer Component Reference

Transformers modify data.
AbsURL

The AbsURL transformer converts a relative file path or URL to an absolute path.

For example, if the input is test.html and the base URL is http://www.example.com, the output is http://www.example.com/test.html.

If the input is an absolute path, the transformer does not alter it.

The following table describes the properties of the AbsURL transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_URL</td>
<td>Defines the base path or URL.</td>
</tr>
</tbody>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
The default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark | A user-defined comment that describes the purpose or action of the component. |

AddEmptyTagsTransformer

The AddEmptyTagsTransformer transformer checks whether all the elements defined in the schema exist in the XML input. If not, it adds empty elements to the XML. This is an XML-to-XML transformer.

The following table describes the properties of the AddEmptyTagsTransformer transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| remark | A user-defined comment that describes the purpose or action of the component. |
| root_element | Defines the root element of the XML. |
AddString

The AddString transformer adds strings before and after the input text.

The following table describes the properties of the AddString transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| pre      | Defines the string to add before the text. |
| post     | Defines the string to add after the text. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

Online Sample

For an online sample, open samples\Projects\Transformers_Example\Transformers_Example.cmw. The first Content anchor in the parser is configured with an AddString transformer.

Base64Decode

The Base64Decode transformer converts the base64 MIME encoding to a binary string.

The following table describes the properties of the Base64Decode transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
<p>| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark | A user-defined comment that describes the purpose or action of the component. |
| tolerance | Controls how the transformer processes whitespace characters or non-base64 sections of its input. You can choose one of the following options:  
- ignore_none. Processes all characters.  
- ignore_non_base64. Processes only base-64 characters. |

**Base64Encode**

The **Base64Encode** transformer converts a binary string to the base64 MIME encoding. This is useful for saving binary data in XML.

The following table describes the properties of the **Base64Encode** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark | A user-defined comment that describes the purpose or action of the component. |

**BidiConvert**

The **BidiConvert** transformer reverses strings that are written in right-to-left (RTL) languages, such as Hebrew and Arabic. The input must be in RTL format. The output is LTR.

The **BidiConvert** transformer operates on Windows where the default language is RTL. For a similar transformer that runs on all platforms, use **hebrewBidi**. The two transformers use slightly different algorithms that occasionally give different results.
The following table described the properties of the **BidiConvert** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

**Note:** This component does not support UTF-8 input encoding. Workaround: Use **hebrewBidi**.

### CDATADecode

The **CDATADecode** transformer decodes a **CDATA** section of an XML document. For example, it converts

```xml
<![CDATA[100 < 200]]>
```

to

```
100 < 200
```

**Note:** If you write the result to XML, the script re-encodes it using the standard XML encoding:

```xml
100 &lt; 200
```

The following table describes the properties of the **CDATADecode** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

### CDATAEncode

The **CDATAEncode** transformer converts a string to a **CDATA** section of an XML document. For example, it converts

```
100 < 200
```
The following table describes the properties of the **CDATAEncode** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| **disabled** | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| **name** | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| **optional** | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see "Failure Handling" on page 289. |
| **remark** | A user-defined comment that describes the purpose or action of the component. |

### ChangeCase

The **ChangeCase** transformer changes a text string to all uppercase, all lowercase, or only the first letter capitalized. This transformer works on English characters. It might fail on some non-English characters. For example, it does not convert lowercase German ß to uppercase SS.

The following table describes the properties of the **ChangeCase** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| **case_type** | Defines the output case. The **case_type** property has the following options:  
- **all_caps**. The output is all uppercase.  
- **all_lower**. The output is all lowercase.  
- **first_cap**. The first letter of the output is uppercase and the rest is lowercase.  
Default is **all_caps**. |
| **disabled** | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| **name** | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| **remark** | A user-defined comment that describes the purpose or action of the component. |

### Online Sample

For an online sample, open **samples\Projects\Transformers_Example\Transformers_Example.cmw**. The third Content anchor in the parser is configured with a **ChangeCase** transformer.
CreateGuid

The **CreateGuid** transformer generates a GUID identifier. The resulting GUID is unique every time this transformer runs.

The GUIDs might have a non-standard format on Linux and UNIX platforms. For a fully UNIX-compatible transformer, use **CreateUUID**. For more information, see "CreateUUID" on page 164.

CreateUUID

The **CreateUUID** transformer generates a UUID identifier that is compatible with Windows, Linux, and UNIX platforms. The resulting UUID is unique every time the transformer runs.

The following table describes the properties of the **CreateUUID** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
  - Selected. Component failure does not cause the parent component to fail.  
  - Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see "Failure Handling" on page 289. |
| remark | A user-defined comment that describes the purpose or action of the component. |

DateFormatICU

The **DateFormatICU** transformer converts a date or time to a format specified by the user.

The following table describes the properties of the **DateFormatICU** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| input_format | Defines the format of the input date, for example, d/M/yy. Type the format or select a data holder that contains the format. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| optional         | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| output_format    | Defines the format of the output date, for example, MM/dd/yyyy. Type the format or select a data holder that contains the format.                                                                          |
| remark           | A user-defined comment that describes the purpose or action of the component.                                                                                                                                |

### Example

Suppose you configure a `DateFormatICU` transformer with:

```java
input_format = "d/M/yyyy"  
output_format = "MM/dd/yyyy"
```

If the input is

```
13/3/05
```

the output is

```
03/13/2005
```

### Supported Formats

The `DateFormatICU` transformer uses the ICU conventions to represent the date and time format. The following table lists the symbols that you can use in the format patterns. For more information, see:

[http://icu.sourceforge.net/apiref/icu4c/classSimpleDateFormat.html](http://icu.sourceforge.net/apiref/icu4c/classSimpleDateFormat.html)

<table>
<thead>
<tr>
<th>Pattern Symbol</th>
<th>Meaning</th>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Era designator</td>
<td>Text</td>
<td>AD</td>
</tr>
<tr>
<td>y</td>
<td>Year</td>
<td>Number</td>
<td>1996</td>
</tr>
<tr>
<td>u</td>
<td>Extended year</td>
<td>Number</td>
<td>-200, meaning 201 BC</td>
</tr>
<tr>
<td>M</td>
<td>Month in year</td>
<td>Text or number</td>
<td>July 07</td>
</tr>
<tr>
<td>d</td>
<td>Day in month</td>
<td>Number</td>
<td>10</td>
</tr>
<tr>
<td>h</td>
<td>Hour in AM/PM (1-12)</td>
<td>Number</td>
<td>12</td>
</tr>
<tr>
<td>H</td>
<td>Hour in day (0-23)</td>
<td>Number</td>
<td>0</td>
</tr>
<tr>
<td>m</td>
<td>Minute in hour</td>
<td>Number</td>
<td>30</td>
</tr>
<tr>
<td>s</td>
<td>Second in minute</td>
<td>Number</td>
<td>55</td>
</tr>
<tr>
<td>Pattern Symbol</td>
<td>Meaning</td>
<td>Type</td>
<td>Examples</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------</td>
<td>-----------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>S</td>
<td>Fractional second</td>
<td>Number</td>
<td>978</td>
</tr>
<tr>
<td>E</td>
<td>Day of week</td>
<td>Text</td>
<td>Tuesday</td>
</tr>
<tr>
<td>e</td>
<td>Day of week (local 1-7)</td>
<td>Number</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>Day in year</td>
<td>Number</td>
<td>189</td>
</tr>
<tr>
<td>F</td>
<td>Day of week in month</td>
<td>Number</td>
<td>2, meaning the 2nd Wednesday in July</td>
</tr>
<tr>
<td>w</td>
<td>Week in year</td>
<td>Number</td>
<td>27</td>
</tr>
<tr>
<td>W</td>
<td>Week in month</td>
<td>Number</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>AM/PM marker</td>
<td>Text</td>
<td>PM</td>
</tr>
<tr>
<td>k</td>
<td>Hour in day (1-24)</td>
<td>Number</td>
<td>24</td>
</tr>
<tr>
<td>K</td>
<td>Hour in AM/PM (0-11)</td>
<td>Number</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>Time zone</td>
<td>Time</td>
<td>Pacific Standard Time</td>
</tr>
<tr>
<td>Z</td>
<td>Time zone (RFC 822)</td>
<td>Number</td>
<td>-0800</td>
</tr>
<tr>
<td>v</td>
<td>Time zone (generic)</td>
<td>Text</td>
<td>Pacific Time</td>
</tr>
<tr>
<td>g</td>
<td>Julian day</td>
<td>Number</td>
<td>2451334</td>
</tr>
<tr>
<td>A</td>
<td>Milliseconds in day</td>
<td>Number</td>
<td>69540000</td>
</tr>
<tr>
<td>' '</td>
<td>The text within single quotes is interpreted as a literal string</td>
<td>Text</td>
<td>'Today is 'dd/MM/yyyy generates output such as Today is 15/03/2005</td>
</tr>
<tr>
<td>' '</td>
<td>Literal single quote</td>
<td>Text</td>
<td>'o''clock' generates the output o'clock</td>
</tr>
</tbody>
</table>

The count of pattern symbols further determines the format:

- For text: Four or more pattern symbols means to use the full form. Fewer than four means to use a short or abbreviated form if it exists. For example, if EEEE produces Monday, EEE produces Mon.
- For numbers: The number of pattern symbols is the minimum number of digits. Shorter numbers are zero-padded. For example, if m produces 6, mm produces 06.
- For years: The two-digit year is yy, and the four-digit year is yyyy. For example, if yy produces 05, yyyy produces 2005.
- For months: If M produces 1, then MM produces 01, MMM produces Jan, and MMMM produces January.

All non-alphabetic characters are interpreted as literals, even if they are not enclosed in single quotes. For example, dd/MM/yyyy HH:mm produces 15/03/2005 13:15.
**Dos96HebToAscii**

The **Dos96HebToAscii** transformer converts the Hebrew 7-bit encoding to the Windows-1255 code page.

**DynamicTable**

The **DynamicTable** component defines a data holder that contains a lookup table. The table is used by the **LookupTransformer** transformer.

The following table describes the properties of the **DynamicTable** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>Defines the data holder that contains the table.</td>
</tr>
</tbody>
</table>

**EbcdicToAscii**

The **EbcdicToAscii** transformer converts EBCDIC to ASCII text.

**EDIFACTValidation**

The **EDIFACTValidation** validator tests whether a source string is a valid EDIFACT message.

The following table describes the properties of the **EDIFACTValidation** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| enabled  | Determines the setting for **param1**.                                                            |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
  - Selected. Component failure does not cause the parent component to fail.  
  - Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| param1   | Determines whether the input is optional. **param1** is named **is_optional** and has only one property, **enabled**.  
  **enabled** has the following options:  
  - Selected. The input data is optional.  
  - Cleared. The input data is mandatory. |
<p>| param2   | Defines an EDI data type. <strong>param2</strong> is named <strong>input_type</strong> and has only one property, <strong>value</strong>. <strong>value</strong> is a hard-keyed string or a data holder. |
| param3   | Defines a range of integers. <strong>param3</strong> is named <strong>minmax_limits</strong> and has only one property, <strong>value</strong>. <strong>value</strong> is a hard-keyed string or a data holder that specifies two integers separated by a hyphen. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>param4</td>
<td>Defines a list of values. <code>param4</code> is named <code>enumerations</code> and has only one property, <code>value</code>. <code>value</code> is a hard-keyed string or a data holder that specifies a comma-separated list of strings or integers.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>value</td>
<td>Defines a value for <code>param1</code>, <code>param2</code>, or <code>param3</code></td>
</tr>
</tbody>
</table>

**Note:** This component is deprecated. The IntelliScript editor displays it for legacy projects. Do not use it in new scripts. **Workaround:** Use other validator components.

### EncodeAsUrl

The `EncodeAsUrl` transformer encodes spaces and special characters as required in a URL. The characters are encoded as a percent sign (%) followed by a hexadecimal number.

For example, the `EncodeAsUrl` transformer converts

```plaintext
http://www.example.com?name=John Doe
```

to

```plaintext
http://www.example.com?name=John%20Doe
```

**Note:** Parenthesis characters are not encoded.

The following table describes the properties of the `EncodeAsUrl` transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: 
- Selected. The script ignores the component. 
- Cleared. The script applies the component. 
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the `name` property to identify the component that caused the event. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

### Online Sample

For an online sample, open `samples\Projects\Transformers_Example\Transformers_Example.cmw`. The fourth content anchor in the parser is configured with an `EncodeAsUrl` transformer.

### Encoder

The `Encoder` transformer converts text from one code page to another.
The following table describes the properties of the **Encoder** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_prefix</td>
<td>Adds a Byte Order Mark (BOM) when the output encoding is UTF-8 or UTF-16.</td>
</tr>
</tbody>
</table>
| disabled       | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared.                     |
| input_code_page| Defines the code page of the input text.                                    |
| name           | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional       | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
  - Selected. Component failure does not cause the parent component to fail.  
  - Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| output_code_page| Defines the code page of the output text.                                  |
| remark         | A user-defined comment that describes the purpose or action of the component. |

**ExternalTransformer**

The **ExternalTransformer** transformer runs a custom transformer that is implemented as a C++ DLL.

The following table describes the properties of the **ExternalTransformer** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled       | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared.                     |
| import_dll     | Specifies a DLL in the externLibs\user folder.                             |
| name           | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional       | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
  - Selected. Component failure does not cause the parent component to fail.  
  - Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark         | A user-defined comment that describes the purpose or action of the component. |
**Note:** This component is deprecated. The IntelliScript editor displays it for legacy scripts. Do not use it in new scripts. Instead, create a custom C or C++ transformer. For more information, see “Developing a Custom Component in C or C++” on page 314.

**FormatNumber**

The **FormatNumber** transformer formats a number by adding a sign, decimal point, leading or trailing zeros, and unit.

The following table describes the properties of the **FormatNumber** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled          | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared.                                                                 |
| insert_decimal_point | Defines the decimal point symbol. You can choose one of the following options:  
  - comma. The decimal is a comma.  
  - none. The output does not have a decimal.  
  - point. The decimal is a period.  
  Default is none.                                                                                          |
| name              | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| number_of_decimals | Pads the decimal part with trailing zeros to the indicated size. Default is 0.                                                                  |
| optional          | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
  - Selected. Component failure does not cause the parent component to fail.  
  - Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark             | A user-defined comment that describes the purpose or action of the component.                                                                    |
| sign              | Determines the sign of the output number. You can choose one of the following options:  
  - un_signed. Deletes a sign if present.  
  - leading_sign. A plus or minus is added to the front of the output number.  
  - trailing_sign. A plus or minus is added after the output number.  
  - negative_sign only. A minus sign is added to the number if the number is negative.  
  - as_in_source. Does not change the input sign.  
  Default is un_signed.                                                                                     |
| size_of_integer_part | Pads the integer part with leading zeros to the indicated size. Default is 0.                                                                     |
| unit_type          | Defines the measurement unit after the number. You can choose one of the following options:  
  - cm  
  - inch  
  - meter  
  - mm  
  - undefined. No unit is added.  
  Default is undefined.                                                                                 |
FromFloat

The **FromFloat** transformer converts a floating point number from binary to an ASCII string representation. The conversion is performed in the input encoding with the input byte-order.

The following table describes the properties of the **FromFloat** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the `name` property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| size     | Determines the size of the input number. You can choose one of the following options:  
- single_precision_32_bit  
- double_precision_64_bit  
Default is single_precision_32_bit. |

**Note:** This component does not support UTF-8 input encoding.

FromInteger

The **FromInteger** transformer converts an integer from binary to an ASCII string representation, in decimal, octal, or hexadecimal. The conversion is performed in the input encoding with the input byte-order.

The following table describes the properties of the **FromInteger** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the `name` property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
### FromPackDecimal

The `FromPackDecimal` transformer converts a number from packed decimal to an ASCII string representation. The conversion is performed in the input encoding with the input byte-order.

The following table describes the properties of the `FromPackDecimal` transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the `name` property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
The default is cleared. For more information about component failure, see "Failure Handling" on page 289. |
| remark | A user-defined comment that describes the purpose or action of the component. |

**Note:** This component does not support UTF-8 input encoding.

### FromSignedDecimal

The `FromSignedDecimal` transformer converts a number from a signed decimal to an ASCII string representation. The conversion is performed in the input encoding with the input byte-order.
The following table describes the properties of the FromSignedDecimal transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled       | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| insert_sign_symbol | Defines the sign of the number. You can choose one of the following options:  
- after. A plus or minus sign is added after the output number.  
- before. A plus or minus sign is added to the front of the output number.  
- no. The output is unsigned.  
Default is no. |
| name           | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional       | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark         | A user-defined comment that describes the purpose or action of the component. |

**Note:** This component does not support UTF-8 input encoding.

**hebrewBidi**

The hebrewBidi transformer reverses a string that is written in right-to-left (RTL) languages, such as Hebrew and Arabic.

The input must be in RTL format. The output is LTR.

**HebrewDosToWindows**

The HebrewDosToWindows transformer converts Hebrew documents from the MS-DOS Hebrew code page to the Windows Hebrew code page.

**HebrewEBCDICOldCodeToWindows**

The HebrewEBCDICOldCodeToWindows transformer converts Hebrew text from EBCDIC to the Windows-1255 code page.

**hebUniToAscii**

The hebUniToAscii transformer converts Hebrew text from Unicode UTF-16 to the Windows-1255 code page.

**hebUtf8ToAscii**

The hebUtf8ToAscii transformer converts Hebrew text from Unicode UTF-8 to the Windows-1255 code page.
The `HtmlEntitiesToASCII` transformer converts HTML entities to plain text. For example, it converts `&copy;` or `&amp;#169;` to the copyright symbol (©).

The following table describes the properties of the `HtmlEntitiesToASCII` transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| remark | A user-defined comment that describes the purpose or action of the component. |

**Supported Entities**

The transformer supports the ISO 8859-1 (Latin-1) entities that are defined in the HTML 4.0 reference, [http://www.w3.org/TR/1998/REC-html40-19980424/smg.html](http://www.w3.org/TR/1998/REC-html40-19980424/smg.html). The supported entities include:

- `&`, `&lt;`, `&gt;`, and `&quot;` (`, `, ` `, respectively)
- Numeric character codes `&0;` to `&255;`
- Entities for Latin-1 characters: `&nbsp;` = non-breaking space, `&copy;` = copyright, etc.

The transformer does not support extended characters, that is, codes greater than 255 or non-Latin-1 characters.

**Output Encoding for Upper-ASCII Characters**

If the transformer output contains upper-ASCII characters, select an output encoding that supports the characters, such as Windows-1252 or UTF-8.

**Note:** Include an encoding attribute in the XML processing instruction. Otherwise, the Developer tool might not be able to display the characters.

**HtmlProcessor**

The `HtmlProcessor` transformer normalizes whitespace according to HTML conventions. It converts any sequence of tabs, line breaks, and space characters to a single space character. This transformer operates on HTML text and any other type of text. You can also use it as a format preprocessor. For more information, see "Format Preprocessor Component Reference" on page 91.

**InjectFP**

The `InjectFP` transformer inserts a decimal point at a specified location in a number. For example, the transformer can convert 12345 to 123.45.
The following table describes the properties of the **InjectFP** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>digits_after_decimal_point</td>
<td>Determines the number of digits after the decimal point.</td>
</tr>
</tbody>
</table>
| disabled                        | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name                            | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional                        | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
The default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark                          | A user-defined comment that describes the purpose or action of the component. |

**InjectString**

The **InjectString** transformer inserts a string into text.

The following table describes the properties of the **InjectString** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled                  | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| injection_place           | Defines the number of characters from the beginning of the text to where the string is inserted. |
| name                      | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| remark                    | A user-defined comment that describes the purpose or action of the component. |
| string_to_inject          | Defines the string to insert.                                              |

**InlineTable**

The **InlineTable** component defines a lookup table in the script. The table is used by the **LookupTransformer** transformer.
The following table describes the properties of the **InlineTable** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>Defines a key and value pair.</td>
</tr>
<tr>
<td>key</td>
<td>Defines a unique input string.</td>
</tr>
</tbody>
</table>
| match_case | Determines whether the key string is case sensitive. You can choose one of the following options:  
- Selected. key is case sensitive.  
- Cleared. key is not case sensitive.  
Default is cleared. |
| table    | Defines a list of Entry components. |
| value    | Defines an output string. |

**JavaTransformer**

The **JavaTransformer** transformer runs a custom transformer that is implemented in Java.

The following table describes the properties of the **JavaTransformer** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| java_class | Defines the path of the Java class. |
| method | Defines the method to run. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark | A user-defined comment that describes the purpose or action of the component. |

**Note:** This component is deprecated. The IntelliScript editor displays it for legacy scripts. Do not use it in new scripts. Instead, create a custom Java transformer. For more information, see “Developing a Custom Component in Java” on page 312.

**LookupTransformer**

The **LookupTransformer** transformer looks up a value in a table.
The following table describes the properties of the `LookupTransformer` transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| look_at  | Defines the type of lookup table used by the transformer. You can choose one of the following options:  
- DynamicTable. The `table` property of the `look_at` property defines a data holder that contains the table. For more information, see “DynamicTable” on page 167.  
- InlineTable. The `table` property of the `look_at` property defines a list of `Entry` components, each of which contains a `key` and a `value`. For more information, see “InlineTable” on page 175.  
- XMLLookupTable. The `xml_file_name` property of the `look_at` property defines the path and file name of an XML file that defines the table. For more information, see “XMLLookupTable” on page 190.  
- [TableName]. A `DynamicTable`, `InlineTable`, or `XMLLookupTable` defined at the global level of the script.  
The default is blank. |
| name     | A descriptive label for the component. This label appears in the log file and the `Events` view. Use the `name` property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
The default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

If you use the same lookup table repeatedly, consider defining an `InlineTable` or an `XMLLookupTable` at the global level of the script. You can then reference the table by name in the `look_at` property.

For example, you can configure a `LookupTransformer` to look up values in the following table:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>George Washington</td>
</tr>
<tr>
<td>2</td>
<td>John Adams</td>
</tr>
<tr>
<td>3</td>
<td>Thomas Jefferson</td>
</tr>
<tr>
<td>4</td>
<td>James Madison</td>
</tr>
</tbody>
</table>

If the input of the transformer is 3, the output is Thomas Jefferson.
Defining an Inline Table

To define an inline table, configure the key-value pairs in the script, as in the following example:

```
<LookUpTable>
  <Entry>
    <key>1</key>
    <value>George Washington</value>
  </Entry>
  <Entry>
    <key>2</key>
    <value>John Adams</value>
  </Entry>
</LookUpTable>
```

Storing an XML Lookup Table in a File

Prepare an XML file conforming with the schema `lookupTableDefinition.xsd`. You can find the schema in the `doc` subdirectory of the installation directory. The following XML document is an example:

```
<?xml version="1.0" encoding="windows-1252" ?>
<lt:LookUpTable xmlns:lt="http://www.Itemfield.com/Engine/V4/lookUpTable" matchCase="false">
  <lt:Entry key="1" value="George Washington" />
  <lt:Entry key="2" value="John Adams" />
</lt:LookUpTable>
```

Creating an XML Lookup Table Dynamically

A transformation can create an XML lookup table at runtime. For example, the transformation might run a secondary parser that generates the XML structure.

The transformation must store the XML string in a multiple-occurrence data holder of type `cde:lookupTable`. Store each key-value pair in an occurrence of the data holder. For example, you might configure a `RepeatingGroup` containing a `WriteValue` action. Each iteration of the `RepeatingGroup` creates an occurrence of the data holder and writes a key-value pair to the occurrence.

Then configure a `LookupTransformer` with the `DynamicTable` option, and specify the data holder.

```
<Variable>
  <value $lookupTable>
    <table>
      <Entry>
        <key>1</key>
        <value>George Washington</value>
      </Entry>
      <Entry>
        <key>2</key>
        <value>John Adams</value>
      </Entry>
    </table>
  </value>$lookupTable
</Variable>
```

NormalizeClosingTags

For XML input, the `NormalizeClosingTags` transformer removes shorthand closing tags from empty elements. It changes `<tag/>` to `<tag></tag>`.

The transformer does not correct incorrect XML. It converts well-formed XML from one style of closing tag to another.
The following table describes the properties of the NormalizeClosingTags transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

**RegularExpression**

The RegularExpression transformer performs a pattern search on the input text. It replaces instances of the pattern with a specified string.

The following table describes the properties of the RegularExpression transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| exp      | Defines a regular expression for the search criterion. |
| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
  - Selected. Component failure does not cause the parent component to fail.  
  - Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
| replacement | Defines the replacement text. |

For example, suppose that a Content anchor retrieves the following text:

```
trans	form
```

You configure the anchor with a RegularExpression transformer that searches for the pattern t.+.s. The pattern means the letter t, followed by one or more of characters, followed by the letter s. You configure the transformer to replace the pattern with the character X.

The pattern matches the substring trans of the input. The transformer replaces the substring and outputs:

```
Xform
er
```
Regular Expression Syntax

A regular expression defines a search pattern according to a standard syntax.


Note: Regex++ does not support locales.

The following table lists some special characters that you can use in regular expressions:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Matches zero or more instances of the preceding character.</td>
<td>*</td>
</tr>
<tr>
<td>?</td>
<td>Matches zero or one instance of the preceding character.</td>
<td>?</td>
</tr>
<tr>
<td>+</td>
<td>Matches one or more instances of the preceding character.</td>
<td>+</td>
</tr>
<tr>
<td>{}</td>
<td>Matches the specified number of instances of the preceding character.</td>
<td>{}</td>
</tr>
<tr>
<td>[]</td>
<td>Matches any of a set of characters.</td>
<td>[ ]</td>
</tr>
<tr>
<td>-</td>
<td>Defines a range of characters inside square brackets.</td>
<td>-</td>
</tr>
<tr>
<td>.</td>
<td>Matches any single character.</td>
<td>.</td>
</tr>
<tr>
<td>^</td>
<td>Matches the start of the input text.</td>
<td>^</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of the input text.</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Matches either of two expressions.</td>
</tr>
<tr>
<td>()</td>
<td>Grouping</td>
<td>()</td>
</tr>
<tr>
<td>\</td>
<td>Escapes one of the other special characters, treating it as a literal character.</td>
<td>\</td>
</tr>
</tbody>
</table>

Preserving Portions of the Original Text

In the `exp` property, you can enclose portions of the regular expression in parentheses. In the `replacement` property, you can use:

- `$0` to identify the entire text that matches the regular expression
- `$1` to identify the substring that matches the first parenthesized portion of the regular expression
- `$2`, `$3`, and so forth, to identify the substrings that match the second, third, etc. parenthesized portions

For example, suppose you set:

```plaintext
exp = abc([0-9]+) (def)
replacement = $1
```
This replaces abc5624def with 5624.

Alternatively, suppose you set:

```
exp = abc([0-9]+) {def}
replacement = $2YX$1
```

This replaces abc5624def with def2YX5624.

RemoveMarginSpace

The RemoveMarginSpace transformer deletes leading and trailing space characters from the text. The following table describes the properties of the RemoveMarginSpace transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

RemoveRtfFormatting

The RemoveRtfFormatting transformer removes RTF formatting instructions from the text. The following table describes the properties of the RemoveRtfFormatting transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

RemoveTags

The RemoveTags transformer removes HTML tags from the input text. It replaces the tags at internal locations in the text with a separator string, such as a space character. It does not insert the separator string at the beginning or end of the text. Adjacent multiple tags are transformed into a single separator.
The following table describes the properties of the **RemoveTags** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled   | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name       | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| optional   | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
The default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark     | A user-defined comment that describes the purpose or action of the component. |
| replace_with | Defines the separator string. Default is " " (space). |

**Replace**

The **Replace** transformer finds and replaces strings in the input text. Leaving the **replace_with** property empty deletes the found text.

The following table describes the properties of the **Replace** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled   | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| find_what  | Defines the text to find. The value is one of the following searcher components:  
- NewlineSearch. Finds a newline character.  
- PatternSearch. Finds text that matches a regular expression.  
- SegmentSearch. Finds a segment from a specified opening marker to a closing marker.  
- TextSearch. Finds a specified string.  
The default is TextSearch. For more information, see the “Searcher Component Reference” on page 144. |
| name       | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| occurrence | Specifies which occurrences to replace: all, first, or last. |
| optional   | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
The default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>replace_with</td>
<td>Defines the replacement string.</td>
</tr>
</tbody>
</table>

#### Online Sample

For an online sample, open samples\Projects\Transformers_Example\Transformers_Example.cmw. The second and fifth Content anchors in the parser are configured with Replace transformers.

---

### Resize

The **Resize** transformer fits the input text to a specified size. It pads or truncates the text as required.

The following table describes the properties of the **Resize** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>align</td>
<td>Defines the text alignment within the resized string. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- left. Padding or trimming is on the right.</td>
</tr>
<tr>
<td></td>
<td>- right. Padding or trimming is on the left.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>padding_character</td>
<td>Defines the padding character. Type the character or select a data holder that contains a character.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>size</td>
<td>Defines the size of the output text. Type an integer or select a data holder that contains an integer.</td>
</tr>
</tbody>
</table>

---

### ReverseTransformer

The **ReverseTransformer** transformer reverses a string. For example, it transforms 1234 to 4321.

The following table describes the properties of the **ReverseTransformer** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
</tbody>
</table>
Property | Description
--- | ---
optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:
- Selected. Component failure does not cause the parent component to fail.
- Cleared. Component failure causes the parent component to fail.
Default is cleared. For more information about component failure, see “Failure Handling” on page 289.

remark | A user-defined comment that describes the purpose or action of the component.

RtfProcessor

The RtfProcessor transformer normalizes RTF code. It is also available as a format preprocessor. For more information, see “Format Preprocessor Component Reference” on page 91.

RtfToASCII

The RtfToASCII transformer converts RTF input to plain text. It removes RTF control words from the text.

The following table describes the properties of the RtfToASCII transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:
- Selected. The script ignores the component.
- Cleared. The script applies the component.
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| remark | A user-defined comment that describes the purpose or action of the component. |

SubString

The SubString transformer returns a substring of the input, starting and ending at specified locations.

The following table describes the properties of the SubString transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>Defines the start location. 0 means to start at the beginning of the input.</td>
</tr>
</tbody>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:
- Selected. The script ignores the component.
- Cleared. The script applies the component.
The default is cleared. |
<p>| end | Defines the end location. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

**ToFloa**t

The **ToFloa**t transformer converts a floating point number from an ASCII string representation to binary. The conversion is performed in the output encoding with the output byte order.

The following table describes the properties of the **ToFloa**t transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>size</td>
<td>Determines the size of the output number. The size property has the following options: - single_precision_32_bit - double_precision_64_bit Default is single_precision_32_bit.</td>
</tr>
</tbody>
</table>

**Note:** This component does not support UTF-8 input encoding.

**ToIntege**r

The **ToIntege**r transformer converts a number from an ASCII string representation to a binary integer. The string input can be decimal, octal, or hexadecimal. The conversion is performed in the output encoding with the output byte order.
The following table describes the properties of the **ToInteger** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| from_base | Defines the base of the input. The to_base property has the following options:  
- hexadecimal. Base 16 using capital letters A-F.  
- lowercase hexadecimal. Base 16 using lowercase letters a-f.  
Default is decimal. |
| name | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark | A user-defined comment that describes the purpose or action of the component. |
| signed | Determines whether the input number has a sign. The to_base property has the following options:  
- Selected. The output number has a sign.  
- Cleared. The output number does not have a sign.  
Default is cleared. |
| size | Defines the size in bytes of the binary representation. The supported values are 1 to 8. |

**Note:** This component does not support UTF-8 input encoding.

**ToPackDecimal**

The **ToPackDecimal** transformer converts a number from an ASCII string representation to packed decimals. The conversion is performed in the output encoding with the output byte order.

The following table describes the properties of the **ToPackDecimal** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
<p>| name | A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the <strong>name</strong> property to identify the component that caused the event. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
| unsigned | Determines whether the packed decimal is signed. You can choose one of the following options:  
- Selected. The packed decimal is unsigned.  
- Cleared. The packed decimal is signed.  
Default is cleared. |

**Note:** This component does not support UTF-8 input encoding.

**TransformationStartTime**

The **TransformationStartTime** transformer outputs the date and time when the transformation started running. The transformer copies the date and time from the **VarSystem** variable and it formats the output according to your specification.

The following table describes the properties of the **TransformationStartTime** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| format   | Defines the format of the date and time. Type the format or select a data holder that contains the format. For more information about the supported formats, see “DateFormatICU” on page 164. |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

**TransformByParser**

The **TransformByParser** transformer runs a parser on its input text. The parser must contain **FindReplaceAnchor** components that mark segments of the text for replacement. When the parser completes execution, the transformer performs the replacements.

The transformer output is the modified text. The script ignores any XML output that the parser generates.
The following table describes the properties of the **TransformByParser** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the *Events* view. Use the name property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| parser   | Defines the name of the parser. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

**Online Sample**

For an online sample, open `samples\Projects\TransformByParser\TransformByParser.cmw`. The sample uses `TransformByParser` to replace every instance of the string `~NL~` with a carriage return followed by a linefeed.

To run the TransformByParser sample:

1. Set `MyTransformByParser` as the startup component.
2. Run the transformer.
3. At the prompt, select the source file `Report.edi`.

   The transformer stores its output in `Results\Transformation of Report.edi`. You can compare the output with the source in Notepad.

**TransformByProcessor**

The **TransformByProcessor** transformer runs a document processor on its input. The output of the transformer is the output of the document processor. For more information, see “Document Processors Overview” on page 66.

The following table describes the properties of the **TransformByProcessor** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the *Events* view. Use the name property to identify the component that caused the event. |
### TransformByService

The **TransformByService** transformer runs a Data Processor transformation service on its input. The output of the transformer is the output of the service.

If you use the transformer to invoke a parser service, the output of the transformer is an XML string.

**Note:** The **TransformByService** transformer supports single-input services. Do not use it with a service that has multiple input ports.

The following table describes the properties of the **TransformByService** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled               | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| disable_automatic_encoding | Determines whether the script applies the input and output encodings that are defined in the service. You can choose one of the following options:  
- Selected. The script ignores the input and output encodings that are defined in the service.  
- Cleared. The script applies the input and output encodings that are defined in the service. |
| name                   | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional               | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
The default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| parameters             | Defines a list of initial values that the script assigns to variables defined in the service. In each element of the list, specify the name of a variable and its value. |
| remark                 | A user-defined comment that describes the purpose or action of the component. |
| service_name           | Defines the name of the service that runs on the input. |
TransformerPipeline

The TransformerPipeline transformer applies a sequence of nested transformers to its input.

The following table describes the properties of the TransformerPipeline transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

XMLLookupTable

The XMLLookupTable component specifies an XML file that contains a lookup table. The table conforms with the lookupTableDefinition.xsd schema in the \doc subdirectory of the installation directory. The table is used by the LookupTransformer transformer.

The following table describes the properties of the XMLLookupTable component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml_file_name</td>
<td>Defines the path and file name of the XML file.</td>
</tr>
</tbody>
</table>

The following XML document is valid against the schema:

```xml
<?xml version="1.0" encoding="windows-1252" ?>
<lt:LookupTable xmlns:lt="http://www.Itemfield.com/Engine/V4/lookupTable"
    matchCase="false">
    <lt:Entry key="1" value="George Washington" />
    <lt:Entry key="2" value="John Adams" />
</lt:LookupTable>
```

If the optional matchCase attribute is true, the key attribute is considered case sensitive.

XSLTTransformer

The XSLTTransformer transformer applies an XSLT transformation to XML input text.

For example, you might use a parser to extract data from an XML document. A Content anchor retrieves a complete, well-formed branch of the XML tree. You can configure the Content anchor with an XSLTTransformer that runs an XSLT transformation on the branch.
The following table describes the properties of the **XSLTTransformer** transformer:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the <code>name</code> property to identify the component that caused the event.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>xslt_file</td>
<td>Defines the path and file name of the XSLT file.</td>
</tr>
</tbody>
</table>
CHAPTER 13

Actions

This chapter includes the following topics:
- Actions Overview, 192
- Standard Action Properties, 193
- Action Component Reference, 194
- Action Subcomponent Reference, 224

Actions Overview

Actions are components that perform operations on data that the script has extracted from a source document. Some examples of the supported actions are:
- Arithmetic computations
- String concatenations
- Submitting forms to a web server
- Activating a secondary parser, serializer, or mapper
- Querying a database

The Data Processor transformation provides many actions, and you can define custom actions.

How Actions Work

An action takes its input from the data holders that are currently available. A single action can have multiple inputs.

If the action is embedded in a parser, the available data holders are the ones that the parser has generated. In a serializer, the data holders are the ones that exist in the input XML, plus any additional data holders that the serializer has generated. For a mapper, the data holders can be in either the input or the output.

The action performs operations on the input and generates output. You can configure many actions to store their output in data holders.

In most actions, the input and output data holders must have simple data types. They must not contain nested elements. A few actions work with data holders that contain nested elements, with multiple-occurrence data holders, or with other special types.

An action can have additional effects, such as writing to a file, updating a database, or submitting data to an external application.
Comparison Between Actions and Transformers

Some actions perform operations that are similar to transformers, for example, modifying a string or querying a database. However, actions differ from transformers in some fundamental ways.

The following table summarizes the differences:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Transformers</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>The input of a transformer is a single string.</td>
<td>The input is implemented by the action. An action can have multiple inputs. The inputs can be data holders.</td>
</tr>
<tr>
<td>Output</td>
<td>The output of a transformer is a string.</td>
<td>The output is implemented by the action. For example, an action can create output data holders.</td>
</tr>
<tr>
<td>Side effects</td>
<td>A transformer has no side effects, other than modifying the input string.</td>
<td>An action can have side effects, such as updating a database.</td>
</tr>
</tbody>
</table>

Defining Actions

Edit the script to define an action. You can insert the actions under the contains line of components such as a Parser, Serializer, Mapper, Group, or RepeatingGroup. Essentially, you can insert actions in any location where you can insert anchors, serialization anchors, or mapper anchors.

The actions run in sequence with the anchors that you specify in the same location. In a parser, you can set the phase property of an action, which determines whether it runs in the initial, main, or final stage of the parsing process. For more information, see “Search Phases” on page 111.

Standard Action Properties

The following table describes standard properties of actions:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options: - Cleared. Take no action. - CustomLog. Write to the user log. - LogError. Write an error message to the engine log. - LogInfo. Write an information message to the engine log. - LogWarning. Write a warning message to the engine log. - NotifyFailure. Send a notification. Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
</tbody>
</table>
### Property Description

**optional**

Determines whether a component failure causes the parent component to fail. You can choose one of the following options:
- Selected. Component failure does not cause the parent component to fail.
- Cleared. Component failure causes the parent component to fail.

Default is cleared. For more information about component failure, see “Failure Handling” on page 289.

**phase**

Determines when the script processes the component. You can choose one of the following options:
- initial. The script processes the component during the initial phase.
- main. The script processes the component during the main phase.
- final. The script processes the component during the final phase.

For more information, see “How a Parser Searches for Anchors” on page 111.

**remark**

A user-defined comment that describes the purpose or action of the component.

### Action Component Reference

Actions perform operations in the system, for example, downloading a file from a remote location or validating a value.

**AddEventAction**

The AddEventAction action adds a message to the event log.

The following table describes the properties of the AddEventAction action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| **disabled** | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| **message** | Defines the message string. |
| **name** | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| **on_fail** | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
**Phase**

Determines when the script processes the component. You can choose one of the following options:
- **Initial.** The script processes the component during the initial phase.
- **Main.** The script processes the component during the main phase.
- **Final.** The script processes the component during the final phase.

For more information, see “How a Parser Searches for Anchors” on page 111.

Default is main.

**Remark**

A user-defined comment that describes the purpose or action of the component.

**Severity**

Determines the severity level of the message. You can choose one of the following options:
- **Notification**
- **Warning**
- **Failure**
- **Fatal error**

Default is notification.

---

**AggregateValues**

The AggregateValues action performs a computation on an aggregate of a multiple-occurrence data holder.

The following table describes the properties of the AggregateValues action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| **aggregation_function** | Determines the function to perform on the aggregate. You can choose one of the following options:  
  - **AllEqual.** Returns true if the values are all the same or false if they are not the same.  
  - **Count.** Returns the number of occurrences of the data holder.  
  - **Join.** Returns a list of all the values, separated by the separator specified in the separator property.  
  - **Sum.** Returns the sum of the values. |
| **AllEqual**      | Defines an option under the aggregation_function property.                                                                                   |
| **Count**         | Defines an option under the aggregation_function property.                                                                                   |
| **data_holder**   | Defines the data holder that stores the output.                                                                                               |
| **disabled**      | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - **Selected.** The script ignores the component.  
  - **Cleared.** The script applies the component.  
  The default is cleared.                                                                 |
| **include_empty_values** | Determines whether the aggregate includes occurrences that contain no data.  
  - **Selected.** The action includes the empty occurrences.  
  - **Cleared.** The action ignores the empty occurrences.  
  Default is selected.                                                                 |
| **Join**          | Defines an option under the aggregation_function property.                                                                                   |
| **name**          | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
Depending on the root_element that you configure, the action can aggregate occurrences at different levels of branching. For example, an XML document has the structure:

```
<Company>
  <Division name="America">
    <Employee>...<Employee>
    <Employee>...<Employee>
    <Employee>...<Employee>
  </Division>
  <Division name="Europe">
    <Employee>...<Employee>
    <Employee>...<Employee>
  </Division>
</Company>
```

If the root_element is Company, and you configure the action to count the occurrences of Employee, the action counts all Employee element that are descendents of Company. The action returns 5.

If the root_element is Division, the action counts the number of Employee occurrences in the Division that the transformation is currently processing. When the action processes America, it returns 3. When it processes Europe, it returns 2.
AppendListItems

The AppendListItems action concatenates the strings in a multiple-occurrence data holder.

The following table describes the properties of the AppendListItems action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| input | Determines the multiple-occurrence data holder for input. The data holder must have a simple data type. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| on_fail | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| output | Determines the data holder that stores the output. The data holder must have a simple data type. |
| phase | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111.  
Default is main. |
| remark | A user-defined comment that describes the purpose or action of the component. |

For more information about preparing the input for this action, see “Mapping to Multiple-Occurrence Data Holders” on page 108.

Example

A source document contains the following space-separated text:

```
HELLO
```

When you parse the document, you want to remove the spaces and store the result in an XML element called Greeting.
Create a multiple-occurrence variable called \textit{VarLetter}. Create several \texttt{Content} anchors that retrieve the individual letters and store them in occurrences of \textit{VarLetter}.

Then, use the \texttt{AppendListItems} action to concatenate the occurrences of \textit{VarLetter} and store the result in the \texttt{Greeting} element. The result is:

\begin{verbatim}
<Greeting>HELLO</Greeting>
\end{verbatim}

**Online Sample**

For an online sample of this action, open the project \texttt{samples\Projects\AppendListItems\AppendListItems.cmw}. The sample uses a \texttt{RepeatingGroup} to store values in a multiple-occurrence variable. It then uses as an \texttt{AppendListItems} action to concatenate the values.

**AppendValues**

The \texttt{AppendValues} action concatenates strings.

The following table describes the properties of the \texttt{AppendValues} action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:
- Selected. The script ignores the component.
- Cleared. The script applies the component.
The default is cleared. |
| input | Determines the list of data holders containing the values to be appended. The data holders must have simple data types. |
| name | A descriptive label for the component. This label appears in the log file and the \texttt{Events} view. Use the \texttt{name} property to identify the component that caused the event. |
| on_fail | The action to take if the component fails. You can choose one of the following options:
- Cleared. Take no action.
- CustomLog. Write to the user log.
- LogError. Write an error message to the engine log.
- LogInfo. Write an information message to the engine log.
- LogWarning. Write a warning message to the engine log.
- NotifyFailure. Send a notification.
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:
- Selected. Component failure does not cause the parent component to fail.
- Cleared. Component failure causes the parent component to fail.
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
<p>| output | Determines the data holder that stores the output. The data holder must have a simple data type. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>phase</td>
<td>Determines when the script processes the component. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- initial. The script processes the component during the initial phase.</td>
</tr>
<tr>
<td></td>
<td>- main. The script processes the component during the main phase.</td>
</tr>
<tr>
<td></td>
<td>- final. The script processes the component during the final phase.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “How a Parser Searches for Anchors” on page 111.</td>
</tr>
<tr>
<td></td>
<td>Default is main.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>skip_unfound_values</td>
<td>Determines whether the action continues when one of the input data holders is missing.</td>
</tr>
<tr>
<td></td>
<td>- Selected. The action continues.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The action fails.</td>
</tr>
<tr>
<td></td>
<td>Default is selected.</td>
</tr>
</tbody>
</table>

### Example

A parser has generated the following XML:

```xml
<Name>
  <First>Ron</First>
  <Last>Lehrer</Last>
  <Name>
```

You can configure an **AppendValues** action that outputs:

```xml
<FullName>Ron Lehrer</FullName>
```

### CalculateValue

Calculates numerical values or concatenates string values.

To calculate numeric values, use the following operators between parameters:

- `+`
- `-`
- `*`
- `/`

You can use parentheses to clarify the numeric expression. You can use variables of the following data types:

- `xs:anyType`
- `xs:anySimpleType`
- numeric data types
- string data types

If the parameters are all numeric data types or numeric strings, CalculateValue performs an arithmetic calculation. Non-integer results are rounded to 14 decimal places.

To concatenate strings, use the plus sign (`+`) operator between parameters and strings.
The following table describes the properties of the **CalculateValue** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| **disabled**  | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared.                                                                                           |
| **expression**| Defines a JavaScript expression. To represent an input parameter, use a dollar sign ($) followed by an integer. To represent a string, enclose it in single quote marks.                                      |
| **failure_action** | Determines the behavior in the event of failure. You can choose one of the following options:  
  - Ignore. The transformation continues.  
  - HaltExecution. The transformation stops.  
  Default is Ignore.                                                                                              |
| **name**      | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event.                      |
| **on_fail**   | The action to take if the component fails. You can choose one of the following options:  
  - Cleared. Take no action.  
  - CustomLog. Write to the user log.  
  - LogError. Write an error message to the engine log.  
  - LogInfo. Write an information message to the engine log.  
  - LogWarning. Write a warning message to the engine log.  
  - NotifyFailure. Send a notification.  
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| **optional**  | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
  - Selected. Component failure does not cause the parent component to fail.  
  - Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289.         |
| **params**    | Defines a list of data holders that contain the input parameters.                                                                                       |
| **phase**     | Determines when the script processes the component. You can choose one of the following options:  
  - initial. The script processes the component during the initial phase.  
  - main. The script processes the component during the main phase.  
  - final. The script processes the component during the final phase.  
  For more information, see “How a Parser Searches for Anchors” on page 111.  
  Default is main.                                                                                                   |
| **remark**    | A user-defined comment that describes the purpose or action of the component.                                                                             |
| **result**    | Determines a data holder that stores the output.                                                                                                          |

**Note:** For more information about the JavaScript syntax that the Data Processor transformation supports, see “EnsureCondition” on page 208. For more information about the precision of `xs:decimal` and `xs:float` values, see “Precision of Numerical Data” on page 96.
Example

A parser has generated the following XML:

```xml
<ItemOrdered>
  <Name>Gizmo</Name>
  <Quantity>100</Quantity>
  <Price>25</Price>
</ItemOrdered>
```

You can use a **CalculateValue** action to generate the output:

```xml
<ItemOrdered>
  <Name>Gizmo</Name>
  <Quantity>100</Quantity>
  <Price>25</Price>
  <Total>2500</Total>
</ItemOrdered>
```

Define the **Name** and **Quantity** elements as input parameters. Specify the JavaScript expression `$1 * $2`, and store the result in the **Total** element.

**Online Sample**

For an online sample of this action, open the project `samples\Projects\CalculateValue\CalculateValue.cmx`. The sample retrieves three numbers from a source document and stores them in variables. It uses a **CalculateValue** action to compute a mathematical function of the numbers.

**CombineValues**

The **CombineValues** action concatenates strings.

The input is a list of data holders and variables. The output is a multiple-occurrence data holder.

If the input is a multiple-occurrence data holder, the **CombineValues** action generates one iteration for each instance of the data holder. On each iteration, the **CombineValues** action combines all of the input data holders and writes the output to one instance of the output data holder.

The following table describes the properties of the **CombineValues** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
<p>| input    | Defines a list of data holders for input. The data holders must have a simple data type. |
| name     | A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the <strong>name</strong> property to identify the component that caused the event. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| on_fail | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| output | Determines the multiple-occurrence data holder where the action stores the output. The data holder must have a simple data type. |
| phase | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111.  
Default is main. |
| remark | A user-defined comment that describes the purpose or action of the component. |

**Example**

In a multiple-occurrence variable called `VarDay`, you have stored the list Monday, Tuesday. In a multiple-occurrence variable called `VarTime`, you have stored morning, afternoon. In a single-occurrence variable called `VarSpace`, you have stored a space character.

Suppose you run `CombineValues` on `VarDay`, `VarSpace`, and `VarTime`, with an output data holder called `DayTime`. The output is:

```
<Datetime>Monday morning</Daytime>
<Datetime>Monday afternoon</Daytime>
<Datetime>Tuesday morning</Daytime>
<Datetime>Tuesday afternoon</Daytime>
```

**Online Sample**

For an online sample of this action, open the project `samples\Projects\CombineValues\CombineValues.cmw`. The sample retrieves lists of days, months, and years from a source document. It uses a `CombineValues` action to generate all possible dates from the lists.

**CreateList**

The `CreateList` action inserts data in a list. The output is a multiple-occurrence data holder containing the list. For more information, see “Multiple-Occurrence Data Holders” on page 103.

Nested in this component, enter the data values.
The following table describes the properties of the **CreateList** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_holder</td>
<td>Defines the multiple-occurrence data holder where the action stores the list. The data holder must have a simple data type.</td>
</tr>
</tbody>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| on_fail | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
The default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| phase | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111.  
The default is main. |
| remark | A user-defined comment that describes the purpose or action of the component. |

**Example**

If the input data values are

```plaintext
Jack  
Jennie  
Larissa
```

the action can create the following output:

```xml
<Name>  
  <First>Jack</First>  
  <First>Jennie</First>  
  <First>Larissa</First>  
</Name>
```

**CustomLog**

The **CustomLog** action can be used as the value of the **on_fail** property. When a failure occurs, the **CustomLog** action runs a serializer that prepares a log message. The system writes the message to a specified output location.
The following table describes the properties of the **CustomLog** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_serializer</td>
<td>Determines the serializer that prepares the log message. Define a serializer in this location, or enter the name of a globally defined serializer.</td>
</tr>
<tr>
<td>output</td>
<td>Determines the output location. The output property has the following options: - OutputDataHolder. Writes to a data holder. - OutputFile. Writes to a file. - OutputPort. Defines the name of an AdditionalOutputPort where the data is written. - ResultFile. Writes to the default results file of the transformation. - StandardErrorLog. Writes to the user log. Default is StandardErrorLog. For more information about these options, see “Action Subcomponent Reference” on page 224 and “Failure Handling” on page 289.</td>
</tr>
</tbody>
</table>

For more information about the **on_fail** property, see “Failure Handling” on page 289.

**DateAddICU**

The **DateAddICU** action increments a date.

The following table describes the properties of the **DateAddICU** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>input_date</td>
<td>Defines the date to be incremented.</td>
</tr>
<tr>
<td>input_format</td>
<td>Defines a string or data holder that defines the date format, for example, <em>dd/MM/yy</em>. For more information, see “DateFormatICU” on page 164.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>num_of_days</td>
<td>Defines a positive or negative integer or a data holder that contains the number of days to add.</td>
</tr>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options: - Cleared. Take no action. - CustomLog. Write to the user log. - LogError. Write an error message to the engine log. - LogInfo. Write an information message to the engine log. - LogWarning. Write a warning message to the engine log. - NotifyFailure. Send a notification. Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
</tbody>
</table>
DateDiffICU

The `DateDiffICU` action computes the difference between two dates.

The following table describes the properties of the `DateDiffICU` action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date1</td>
<td>Defines a string or data holder that defines the first date.</td>
</tr>
<tr>
<td>date2</td>
<td>Defines a string or data holder that defines the second date.</td>
</tr>
<tr>
<td>date_format1</td>
<td>Defines a string or data holder that defines the format of the first date, example, (dd/MM/yyyy). If you omit the format, the system default is used. For more information, see “DateFormatICU” on page 164.</td>
</tr>
<tr>
<td>date_format2</td>
<td>Defines a string or data holder that defines the format of the second date.</td>
</tr>
</tbody>
</table>
| disabled   | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
  The default is cleared. |
| name       | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| on_fail    | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional   | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289.
**Property** | **Description**
--- | ---
output | Defines the data holder that stores the result.
phase | Determines when the script processes the component. You can choose one of the following options:
- initial. The script processes the component during the initial phase.
- main. The script processes the component during the main phase.
- final. The script processes the component during the final phase.
For more information, see “How a Parser Searches for Anchors” on page 111. Default is main.
remark | A user-defined comment that describes the purpose or action of the component.

downloadFileToDataHolder

The **DownloadFileToDataHolder** action downloads a file from a web server and stores its content in a data holder. The action converts symbols to XML entities.

The following table describes the properties of the **DownloadFileToDataHolder** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:
- Selected. The script ignores the component.
- Cleared. The script applies the component.
The default is cleared. |
| file_url | Determines a data holder that stores the URL of the file. |
| name | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| on_fail | The action to take if the component fails. You can choose one of the following options:
- Cleared. Take no action.
- CustomLog. Write to the user log.
- LogError. Write an error message to the engine log.
- LogInfo. Write an information message to the engine log.
- LogWarning. Write a warning message to the engine log.
- NotifyFailure. Send a notification.
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:
- Selected. Component failure does not cause the parent component to fail.
- Cleared. Component failure causes the parent component to fail.
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| output | Determines the data holder that stores the downloaded content. |
## Property Description

### phase
Determine when the script processes the component. You can choose one of the following options:
- initial. The script processes the component during the initial phase.
- main. The script processes the component during the main phase.
- final. The script processes the component during the final phase.

For more information, see “How a Parser Searches for Anchors” on page 111.
Default is main.

### remark
A user-defined comment that describes the purpose or action of the component.

### DumpValues

The **DumpValues** action is a debugging tool. It writes data to a `<DumpValues>...</DumpValues>` element. Define the data holders that you want to dump by nesting them as child components.

The following table describes the properties of the **DumpValues** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| on_fail | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
The default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| output | Determines the data holder that stores the result. You can choose one of the following options:  
- OutputFile  
- ResultFile  
- StandardErrorLog  
The default is ResultFile. |
| phase | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  

For more information, see “How a Parser Searches for Anchors” on page 111.  
The default is main. |
| remark | A user-defined comment that describes the purpose or action of the component. |
EnsureCondition

The **EnsureCondition** action evaluates a Boolean JavaScript expression. If the expression is *false*, the action fails.

The following table describes the properties of the **EnsureCondition** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| condition | Defines a JavaScript expression for evaluation. In the expression, refer to the parameters defined in **params** with a dollar sign ($) followed by an integer. For example, the following expression checks whether the first parameter has the value *Ron Lehrer*:

```javascript
$1 == "Ron Lehrer"
```

| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:
- Selected. The script ignores the component.
- Cleared. The script applies the component.
  
  The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| on_fail | The action to take if the component fails. You can choose one of the following options:
- Cleared. Take no action.
- CustomLog. Write to the user log.
- LogError. Write an error message to the engine log.
- LogInfo. Write an information message to the engine log.
- LogWarning. Write a warning message to the engine log.
- NotifyFailure. Send a notification.
  
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| params | Defines a list of data holders. |
| phase | Determines when the script processes the component. You can choose one of the following options:
- initial. The script processes the component during the initial phase.
- main. The script processes the component during the main phase.
- final. The script processes the component during the final phase.
  
  For more information, see “How a Parser Searches for Anchors” on page 111. Default is main. |
| remark | A user-defined comment that describes the purpose or action of the component. |

**Standard JavaScript Syntax**

The JavaScript processor supports standard JavaScript expressions containing the following features.

- The unary and binary operators:
  
  ```javascript
  {} + - * / % == != < <= > >= & & ||
  ```

- The ternary `?:` operator.

- The following methods:
  
  ```javascript
  charAt
  indexOf
  lastIndexOf
  length
  ```
The JavaScript processor does not support features such as the following:

- The unary and binary operators:
  
  ```javascript
  ++ -- typeof void >> >>> << === !== & ~ | ^
  ```

- Assignment operators:

  ```javascript
  = += -= *= /= >>= >>>= <<= %= |= ^=
  ```

- The comma operator (,,).

- The values NaN, null, infinity, or -0 (negative 0).

- Data types other than string, number, and boolean.

- The Date object.

- The equalsIgnoreCase function.

### JavaScript Extensions

The JavaScript processor implements the following methods that are not defined in standard JavaScript. You can use these extensions in any location where the script accepts a JavaScript expression.

Most of the functions are JavaScript implementations of transformers or actions.

```javascript
extra.sum(number1, number 2, ...)
```

Returns the sum of the parameters.

```javascript
extra.allSame(param1, param2, ...)
```

Returns true if all the parameters have the same value.

```javascript
lookup.<lookup_name>({key})
```

This function accesses a global lookup table by name.

In the script, define a global InlineTable or XMLLookupTable. Then, in a JavaScript expression, you can access the table. For example, if you define a global InlineTable called US Presidents, lookup.US Presidents(1) returns George Washington.

For more information, see "LookupTransformer" on page 176.

```javascript
extra.formatDate(date, input_format, output_format)
```

Formats a date or time. For more information, see “DateFormatICU” on page 164.
The **ExcludeItems** action removes specified values from a multiple-occurrence data holder. The data holder type must be simple. To exclude specific strings from the data holder, define them as child components. For more information, see “Multiple-Occurrence Data Holders” on page 103.
The following table describes the properties of the **Excludeltems** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_holder</td>
<td>Defines a multiple-occurrence data holder.</td>
</tr>
</tbody>
</table>
| disabled   | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared.                                       |
| name       | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| on_fail    | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional   | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| phase      | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111.  
Default is main. |
| remark     | A user-defined comment that describes the purpose or action of the component.                                                               |

**Map**

The **Map** action copies a value from one data holder to another.

The following table describes the properties of the **Map** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled   | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared.                                       |
<p>| name       | A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the <strong>name</strong> property to identify the component that caused the event. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>on_fail</strong></td>
<td>The action to take if the component fails. You can choose one of the following options: - Cleared. Take no action. - CustomLog. Write to the user log. - LogError. Write an error message to the engine log. - LogInfo. Write an information message to the engine log. - LogWarning. Write a warning message to the engine log. - NotifyFailure. Send a notification. Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td><strong>optional</strong></td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td><strong>phase</strong></td>
<td>Determines when the script processes the component. You can choose one of the following options: - initial. The script processes the component during the initial phase. - main. The script processes the component during the main phase. - final. The script processes the component during the final phase. For more information, see “How a Parser Searches for Anchors” on page 111. Default is main.</td>
</tr>
<tr>
<td><strong>remark</strong></td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td><strong>source</strong></td>
<td>Determines the source data holder.</td>
</tr>
<tr>
<td><strong>source_default</strong></td>
<td>Determines the default source data holder.</td>
</tr>
<tr>
<td><strong>target</strong></td>
<td>Determines the destination data holder.</td>
</tr>
<tr>
<td><strong>transformers</strong></td>
<td>Defines a sequence of transformers that modify the value. Do not assign this property if the source and destination are complex XML elements.</td>
</tr>
<tr>
<td><strong>validators</strong></td>
<td>Defines a list of validators applied to the source data. For more information, see “Validators” on page 292.</td>
</tr>
</tbody>
</table>

When you copy a data holder that has a simple data type, the source and destination must have compatible data types. The action can apply transformers to the copied value.

When you copy a multiple-occurrence data holder that has a simple type, and the action is not located within an iterating component such as a RepeatingGroup, the action copies all the occurrences of the data holder.

When you copy a data holder that has a complex type, the source and destination must have identical internal structures and identical data types. The action copies the nested elements and attributes.

**Online Sample**

For an online sample of this action, open the project samples\Projects\CopyValue\CopyValue.cmw. The sample uses a Map action to copy a complex element that contains an attribute and nested elements.

**Notify**

The Notify action triggers a notification. Use it to insert a warning message in the transformation output.
The following table describes the properties of the Notify action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notify | Defines a notification. You can choose one of the following options:  
- MismatchIds. The record and subelement IDs partially match.  
- StructureBelowMinOccurs. There are fewer matching records of the subelement than defined in minOccurs.  
- StructureExceedsMaxOccurs. There are more matching records of the subelement than defined in maxOccurs.  
- StructureOutOfSequence. The records match the subelements but not in the required sequence.  
- UnexpectedRecord. The records match the subelements, but not in the required hierarchy.  
- UnrecognizedRecord. No subelement matches any of the record identifiers.  
- XsdValidation. The input does not match the requirements of the schema.  
- User-defined Notification or NotificationGroup component. User-defined message.  
For more information, see “Notifications” on page 143. |
| phase | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111.  
Default is final. |
| value | Defines a value to assign to the VarNotificationDetails/Value variable. A NotificationHandler can include the value in its output. |

ResetVisitedPages

The ResetVisitedPages action clears the list of visited pages of specified secondary parsers. It allows multiple visits to the same page, even if reject_recurring_pages is selected. Use this action to post different input data to the same web page. This action is used with the reject_recurring_pages property of a Parser component.

The following table describes the properties of the ResetVisitedPages action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
### Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| on_fail  | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |

| parsers  | Defines a list of parsers. The list of visited pages of each parser is reset. |

| phase    | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111. Default is main. |

| remark   | A user-defined comment that describes the purpose or action of the component. |

---

**RunMapper**

The **RunMapper** action runs a mapper as a subcomponent of a parser, mapper, or serializer.

The following table describes the properties of the **RunMapper** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |

| input    | Defines a data holder that contains XML text on which to run the mapper.  
The data holder must have a simple data type such as `xs:string`. The value of the string can be XML text of any complexity. For more information about how to run a mapper on a data holder that has a complex type, see “EmbeddedMapper” on page 249.  
If you omit this property, the mapper uses the data holders available in the scope of the action. For example, if the action is nested in a parser, the mapper runs on the output of the parser. If the action is within a **Group**, it runs on the output of the **Group**. |

| mapper   | Defines the mapper. Select the name of an existing **Mapper** component or define a **Mapper** component at this location of the script. For more information, see “Mapper Component Reference” on page 246. |

| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
### on_fail

The action to take if the component fails. You can choose one of the following options:
- Cleared. Take no action.
- CustomLog. Write to the user log.
- LogError. Write an error message to the engine log.
- LogInfo. Write an information message to the engine log.
- LogWarning. Write a warning message to the engine log.
- NotifyFailure. Send a notification.

Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.

### optional

Determines whether a component failure causes the parent component to fail. You can choose one of the following options:
- Selected. Component failure does not cause the parent component to fail.
- Cleared. Component failure causes the parent component to fail.

Default is cleared. For more information about component failure, see “Failure Handling” on page 289.

### phase

Determines when the script processes the component. You can choose one of the following options:
- initial. The script processes the component during the initial phase.
- main. The script processes the component during the main phase.
- final. The script processes the component during the final phase.

For more information, see “How a Parser Searches for Anchors” on page 111. Default is main.

### remark

A user-defined comment that describes the purpose or action of the component.

---

### RunParser

The RunParser action runs a secondary parser. The output of RunParser is appended to the output of the main component that activated it, such as a parser or serializer.

Use the RunParser action to follow the links in an HTML file and run a secondary parser on the link destinations. In a serializer, you can use it to parse bits of unstructured data in the input.

Note the following difference between the RunParser action and the EmbeddedParser anchor:

- RunParser parses a new source.
- EmbeddedParser parses a section of an existing source.

The following table describes the properties of the RunParser action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>exclude_strings</td>
<td>Defines the strings that must be absent from the input_source. If a specified string is present, the RunParser action does not access the source or activate the secondary parser.</td>
</tr>
<tr>
<td>include_strings</td>
<td>Defines the strings that must be present in the input_source value. If a specified string is missing, the RunParser action does not access the source or activate the secondary parser.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| input_source      | Defines a data holder that contains one of the following objects:  
- If `input_source_as_text` is selected, `input_source` contains a string.  
- If `input_source_as_text` is cleared, `input_source` contains the path and file name of the input document.  
Default is the `VarLinkURL` system variable. |
| input_source_as_text | Determines the type of data in the `input_source` data holder.  
- Selected. `input_source` contains a text string.  
- Cleared. `input_source` contains a file path.  
Default is cleared. |
| name              | A descriptive label for the component. This label appears in the log file and the Events view. Use the `name` property to identify the component that caused the event. |
| next_parser       | Defines the name of the parser to run. A recursive call to the same parser is permitted. |
| on_fail           | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional          | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| phase             | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111. Default is main. |
| pre_processor     | Defines a document processor to apply to the source after the document processor defined in the associated `AdditionalInputPort > pre_processor`. |
| remark            | A user-defined comment that describes the purpose or action of the component. |
| retries           | Defines the number of times to retry if the request fails. Default is 0. |
| seconds_to_wait   | Defines the interval in seconds between retries. Default is 60. |

**Example**

An HTML file has a link to a second file. A **Content** anchor stores the file path of the link destination in the `VarLinkURL` system variable. The **RunParser** action accesses the destination file and runs a secondary parser on it.
In another example, the main parser contains an Alternatives anchor that selects a secondary parser according to text in the source document. For more information, see the “Alternatives” on page 118.

RunSerializer

The RunSerializer action runs a serializer as a subcomponent of a parser, mapper, or serializer. The output of the serializer is stored in a data holder.

The following table describes the properties of the RunSerializer action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>input</td>
<td>Defines a data holder that contains XML text on which the serializer runs. The data holder must have a simple data type such as xs:string. The value of the string can be XML text of any complexity. For more information about how to run a serializer on a data holder that has a complex type, see “EmbeddedSerializer” on page 237. If you omit this property, the serializer uses the data holders available in the scope of the action. For example, if the action is nested in a parser, the serializer runs on the output of the parser. If the action is within a Group, it runs on the output of the Group.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options: - Cleared. Take no action. - CustomLog. Write to the user log. - LogError. Write an error message to the engine log. - LogInfo. Write an information message to the engine log. - LogWarning. Write a warning message to the engine log. - NotifyFailure. Send a notification. Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>output</td>
<td>Defines a data holder for the serializer output.</td>
</tr>
<tr>
<td>phase</td>
<td>Determines when the script processes the component. You can choose one of the following options: - initial. The script processes the component during the initial phase. - main. The script processes the component during the main phase. - final. The script processes the component during the final phase. For more information, see “How a Parser Searches for Anchors” on page 111. Default is main.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>serializer</td>
<td>Defines a serializer. Select the name of an existing Serializer component or define a Serializer at this location in the script. For more information, see “Serializer Component Reference” on page 231.</td>
</tr>
</tbody>
</table>
Online Sample

For an online sample of this action, open the project `samples\Projects\RunSerializer\RunSerializer.cmw`.

To observe how the sample works, set `MainParser` as the startup component and run it. `MainParser` contains a `RepeatingGroup` that parses pairs of names and stores them in variables. After each iteration, the `RepeatingGroup` executes a `RunSerializer` action that concatenates the variables with some predefined text. The action stores its output in an XML element that is added to the parser output.

RunXMap

The `RunXMap` action runs an XMap object as a subcomponent of a parser, a mapper, or a serializer.

The following table describes the properties of the `RunXMap` action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| input | The input for the XMap object. You can choose one of the following options:  
- Data Holder. Use a complex type that must match the XMap input type.  
- XML string. The XML string is the input, or a simple type data holder such as `xs:string`. The XML root type must match the XMap input type.  
If you do not use this property, the XMap object uses its default input. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the `name` property to identify the component that caused the event. |
| on_fail | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| output | A data holder of a complex type that must match the XMap output type.  
If you omit this property, the XMap object writes to its default output definition. |
| phase | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111.  
Default is main. |
### SetValue

The **SetValue** action places predefined content into a data holder. If the data holder is a single-occurrence data holder, existing content is replaced. If the data holder is a multiple-occurrence data holder, the defined content is appended to the end.

The following table describes the properties of the **SetValue** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data_holder</strong></td>
<td>Defines a data holder that receives the output.</td>
</tr>
</tbody>
</table>
| **disabled**    | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignites the component.  
  - Cleared. The script applies the component.
  The default is cleared. |
| **name**       | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| **on_fail**    | The action to take if the component fails. You can choose one of the following options:  
  - Cleared. Take no action.  
  - CustomLog. Write to the user log.  
  - LogError. Write an error message to the engine log.  
  - LogInfo. Write an information message to the engine log.  
  - LogWarning. Write a warning message to the engine log.  
  - NotifyFailure. Send a notification.
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| **optional**   | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
  - Selected. Component failure does not cause the parent component to fail.  
  - Cleared. Component failure causes the parent component to fail.
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| **phase**      | Determines when the script processes the component. You can choose one of the following options:  
  - initial. The script processes the component during the initial phase.  
  - main. The script processes the component during the main phase.  
  - final. The script processes the component during the final phase.
  For more information, see “How a Parser Searches for Anchors” on page 111.  
  Default is main. |
| **quote**      | Defines a string for the content of the data holder.                       |
| **remark**     | A user-defined comment that describes the purpose or action of the component. |
| **transformers** | Defines a list of transformers that are applied to the content before it is saved in the data holder. |
The **Sort** action sorts the occurrences of a multiple-occurrence data holder. The output the original content of the data holder. The sort is case sensitive.

If you run the action on an XML element that contains attributes or nested elements, you can use them as sort keys.

The following table describes the properties of the **Sort** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>by_fields</td>
<td>Defines a list of sort keys in decreasing order of precedence. For each field, select the data holder and an ascending or descending sort. You can select the multiple-occurrence data holder itself, or any of its nested elements or attributes. To sort numerically, a sort key must have a numerical data type such as <code>xs:integer</code>.</td>
</tr>
</tbody>
</table>
| disabled    | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| name        | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| on_fail     | The action to take if the component fails. You can choose one of the following options:  
  - Cleared. Take no action.  
  - CustomLog. Write to the user log.  
  - LogError. Write an error message to the engine log.  
  - LogInfo. Write an information message to the engine log.  
  - LogWarning. Write a warning message to the engine log.  
  - NotifyFailure. Send a notification.  
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| phase       | Determines when the script processes the component. You can choose one of the following options:  
  - initial. The script processes the component during the initial phase.  
  - main. The script processes the component during the main phase.  
  - final. The script processes the component during the final phase.  
  For more information, see “How a Parser Searches for Anchors” on page 111.  
  Default is main. |
| recurring_element | Defines a multiple-occurrence data holder to be sorted. |
| remark      | A user-defined comment that describes the purpose or action of the component. |

**Limitation**

You cannot use the **Sort** action if a **Key** is defined on the multiple-occurrence data holder. For more information, see “Overview of Locators, Keys, and Indexing” on page 253.

**ValidateValue**

The **ValidateValue** action validates XML data according to a set of user-defined rules. If the data violates the rules, the action saves a validation report in a data holder.
The input of the action is a data holder. If the data holder is the root of an XML branch, the action analyzes the entire branch.

The following table describes the properties of the ValidateValue action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>errors_found</td>
<td>Defines a data holder that counts the number of validation rule violations in the input.</td>
</tr>
<tr>
<td>errors_output</td>
<td>Defines a data holder where the action stores the XML validation error report. If the data holder has the type xs:string, the output is a string containing XML tags. If the data holder has the type cde:validationErrors, the output is a structure containing nested data holders.</td>
</tr>
<tr>
<td>input</td>
<td>Defines the input data holder that the action analyzes for conformity with the VRL validation rules.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>phase</td>
<td>Determines when the script processes the component. You can choose one of the following options: - initial. The script processes the component during the initial phase. - main. The script processes the component during the main phase. - final. The script processes the component during the final phase. For more information, see “How a Parser Searches for Anchors” on page 111.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

WriteValue

The WriteValue action writes the value of a data holder to a location such as a file or to a string-type data holder.

If the input data holder is an XML element, the action writes both the element and any nested elements and attributes.

The following table describes the properties of the WriteValue action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>input</td>
<td>Defines the data holder to write from.</td>
</tr>
</tbody>
</table>
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td>A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the <strong>name</strong> property to identify the component that caused the event.</td>
</tr>
</tbody>
</table>
| **no_tags** | Determines whether the result is surrounded by XML tags. You can choose one of the following options:  
- Selected. XML tags are omitted. This is appropriate only if **input** is a simple data holder, containing no nested elements or attributes.  
- Cleared. The result is surrounded by XML tags. This is the default.  
Default is cleared. |
| **on_fail** | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| **optional** | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| **output** | Defines the output location. You can choose one of the following options:  
- OutputDataHolder. Writes to a data holder.  
- OutputFile. Writes to a file.  
- OutputPort. Defines the name of an **AdditionalOutputPort** where the data is written.  
- ResultFile. Writes to the default results file of the transformation.  
- StandardErrorLog. Writes to the user log. For more information, see “Failure Handling” on page 289.  
Default is ResultFile. For more information about these options, see “Action Subcomponent Reference” on page 224. |
| **phase** | Determines when the script processes the component. You can choose one of the following options:  
- initial. The script processes the component during the initial phase.  
- main. The script processes the component during the main phase.  
- final. The script processes the component during the final phase.  
For more information, see “How a Parser Searches for Anchors” on page 111.  
Default is main. |
| **remark** | A user-defined comment that describes the purpose or action of the component. |
| **transformers** | Defines a list of transformers that modify the value before writing. The input to the transformers is the complete **input** data holder, including XML tags. |

### Online Sample

For an online sample of this action, open the project `samples\Projects\Splitter\Splitter.cmw`.

The sample demonstrates how to split a file into two files. A parser uses a **RepeatingGroup** to retrieve the records of an HL7 file. It uses a **Map** action to create unique filenames for each record, and a **WriteValue** action to write the records to the files. The output files, `MyOutput1.txt` and `MyOutput2.txt`, are stored in the Results folder of the project.

**Note:** You can use a streamer to split large inputs. For more information, see “Streamers Overview” on page 269.
**XSLTMap**

The **XSLTMap** action runs an XSLT transformation. The input and output are branches of an XML document. They can be the output of a parser or the input of a serializer.

The following table describes the properties of the **XSLTMap** action:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>input</td>
<td>Defines a data holder that contains the XML element to transform.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options: - Cleared. Take no action. - CustomLog. Write to the user log. - LogError. Write an error message to the engine log. - LogInfo. Write an information message to the engine log. - LogWarning. Write a warning message to the engine log. - NotifyFailure. Send a notification. Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>output</td>
<td>Defines a data holder to store the output.</td>
</tr>
<tr>
<td>phase</td>
<td>Determines when the script processes the component. You can choose one of the following options: - initial. The script processes the component during the initial phase. - main. The script processes the component during the main phase. - final. The script processes the component during the final phase. For more information, see “How a Parser Searches for Anchors” on page 111. Default is main.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>xslt_file</td>
<td>Defines the XSLT file.</td>
</tr>
</tbody>
</table>

**Example**

Suppose that the following XML is the result of a parser:

```xml
<Person>
  <First>Ron</First>
  <Last>Lehrer</Last>
</Person>
```
With an appropriate XSLT file, you can use the XSLTMap action to convert this to:

```xml
<Person Name="Lehrer, Ron" />
```

## Action Subcomponent Reference

Action subcomponents serve as the values of certain properties of actions.

### OutputDataHolder

The **OutputDataHolder** subcomponent directs the output to a data holder. It is used in the `output` property of the **WriteValue** action.

The following table describes the properties of the **OutputDataHolder** subcomponent:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_holder</td>
<td>Defines the output data holder.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a sequence of transformers that modify the stream before writing.</td>
</tr>
</tbody>
</table>

### OutputFile

The **OutputFile** subcomponent directs the output to a file. It is used in the `output` property of the **DumpValues** and **WriteValue** actions.

The following table describes the properties of the **OutputFile** subcomponent:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| append | Determines whether the data is appended to the existing content of the file. You can choose one of the following options:  
- Selected. The data is appended to the existing content.  
- Cleared. The data overwrites the existing content.  
Default is cleared. |
| file | Defines a string or data holder that defines the path and file name. The path can be absolute or relative. If the path is relative, the script resolves the path relative to the output folder of the transformation. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| remark | A user-defined comment that describes the purpose or action of the component. |
ResultFile

The ResultFile subcomponent specifies that the output is the normal output file of a Data Processor transformation. It is used in the DumpValues and WriteValue actions.

StandardErrorLog

The StandardErrorLog subcomponent specifies that the output is the user log.
Serializers

This chapter includes the following topics:
- Serializers Overview, 226
- Serialization Anchors, 229
- Standard Serializer Properties, 231
- Serializer Component Reference, 231
- Serialization Anchor Component Reference, 232

Serializers Overview

A serializer converts an XML file to an output document in any format. Serialization is the opposite of parsing. For example, the output of a serializer can be a text document, an HTML document, or even another XML document.

You can create a serializer by the following methods:
- By inverting the configuration of an existing parser
- By editing the script and inserting a Serializer component

You can combine these methods. You can invert a parser and edit the script of the resulting serializer.

It is usually easier to create a serializer than a parser because the XML input is completely structured. The structure makes it easy to identify the required data and write it, in a sequential procedure, to the output. A parser, in contrast, may need to process unstructured or semi-structured input, a task that is often complex.

The main components nested in a serializer are serialization anchors. The function of the serialization anchors is to identify the XML data and write it to the output. Serialization anchors are analogous to the anchors in a parser, except that they work in the opposite direction.

Creating a Serializer by Inverting a Parser

You can create a serializer by inverting the configuration of an existing parser. For example, if you have a parser that transforms tab-delimited text to XML, you can create a serializer that transforms the XML to tab-delimited text.

1. Open a script object in the IntelliScript editor.
2. Right-click a parser, and then click Create Serializer.

The serializer appears in the script. The name of the serializer is derived from that of the parser, with the suffix _serializer. For example, if you create a serializer from Parser1, the serializer is called Parser1_serializer.

The Data Processor transformer stores the serializer in a new TGP script file, which has a name such as Parser1_auto_generated_serializer.tgp.
3. Test the serializer.
   If necessary, edit the script. For more information, see “Troubleshooting an Auto-Generated Serializer” on page 228.

Online Sample
For an example of an auto-generated serializer, open samples\Projects\Serialization\TabDelimited\TabDelimited.cmw.

To run the sample:
1. Set MyHL7Parser as the startup component, and run it. This generates an output file Results\output.xml.
2. Now set MyHL7Parser_serializer as the startup component, and run it. At the prompt, browse to Results\output.xml as the input. The original input file is regenerated.

A variant of this project is in samples\Projects\Serialization\HL7\HL7.cmw. You can generate the serializer yourself and try the above experiment.

Controlling How the Create Serializer Command Works
When you run the Create Serializer command, the Developer tool converts the Content anchors of the parser to ContentSerializer serialization anchors.

By default, the command converts all other text in the example source to StringSerializer serialization anchors. If the other text is boilerplate content, the output of the serializer contains all the boilerplate that was in the original example source.

For example, suppose the parser runs on tab-delimited source documents having the following structure:

```
Name (first and last):<tab>Ron Lehrer
```

Assume that the anchors are defined in the following way:

<table>
<thead>
<tr>
<th>Source text</th>
<th>Anchor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Marker</td>
</tr>
<tr>
<td>(first and last):&lt;tab&gt;</td>
<td>Not marked as an anchor</td>
</tr>
<tr>
<td>Ron Lehrer</td>
<td>Content</td>
</tr>
</tbody>
</table>

The XML output of the parser is:
```
<FullName>Ron Lehrer<FullName>
```

Now, generate a serializer from this parser, and run the serializer on the following input:
```
<FullName>Larissa Chan<FullName>
```

The output of the serializer is:
```
Name (first and last):<tab>Larissa Chan
```

Serialization Mode
The example source might contain text that you do not want in the serializer output. In that case, you can modify the behavior of the Create Serializer command in a way that does not generate the StringSerializer serialization anchors.
To do this, set the `serialization_mode` property of the **Parser** component. The possible values of the `serialization_mode` are explained in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full</strong></td>
<td>The <strong>Create Serializer</strong> command copies the non-XML text to the serializer configuration. This is the default behavior.</td>
</tr>
<tr>
<td><strong>Outline</strong></td>
<td>The <strong>Create Serializer</strong> command copies only the delimiters of the non-XML text to the serializer configuration. Under the <strong>Outline</strong> option, you can select the <code>use_markers</code> option. This causes the Create Serializer command to copy the content of the <strong>Marker</strong> anchors but only the delimiters of other non-XML text.</td>
</tr>
</tbody>
</table>

The following table illustrates the results of the `serialization_mode` settings:

<table>
<thead>
<tr>
<th>serialization_mode</th>
<th>Behavior</th>
<th>Sample Serializer Output</th>
</tr>
</thead>
</table>
| outline With `use_markers` cleared | The **Create Serializer** command converts:  
- **Content** anchors to **ContentSerializer** serialization anchors  
- The delimiters of other text in the example source to **StringSerializer** serialization anchors | `<tab>Larissa Chan` |
| outline With `use_markers` selected | The **Create Serializer** command converts:  
- **Content** anchors to **ContentSerializer** serialization anchors  
- The complete text of **Marker** anchors to **StringSerializer** serialization anchors  
- The delimiters of other text in the example source to **StringSerializer** serialization anchors | Name<tab>Larissa Chan |
| full | The **Create Serializer** command converts:  
- **Content** anchors to **ContentSerializer** serialization anchors  
- All other text in the example source to **StringSerializer** serialization anchors | Name (first and last)<tab>Larissa Chan |

**Troubleshooting an Auto-Generated Serializer**

Often, you can use an automatically generated serializer directly. If required, you can edit the auto-generated serializer to correct any limitations or problems that you find in it.

The following paragraphs list some typical circumstances under which you need to edit the serializer, and the suggested editing steps.

**Root Tag**

In the XML Generation tab of the Data Processor transformation settings, you can configure the script to wrap a root element around the root element defined in the output schema.

If you then use the output XML as the input of an auto-generated serializer, you must set the `root_tag` property of the serializer to the name of the root element defined in the XML Generation settings.

**Variables**

If the parser uses a variable to store intermediate results, an auto-generated serializer may fail. To solve the problem, review the serializer logic, and remove the variable if necessary.
**Additional Components**

The Create Serializer command inverts the anchors of a parser. It does not invert components such as document processors, transformers, or actions.

For example, suppose that a parser uses a `PdfToTxt_4` document processor to convert PDF source documents to text. The parser contains anchors that transform the text to XML.

The auto-generated serializer transforms the XML back to text. It does not convert the text to PDF. To obtain PDF output, edit the serializer and insert an `XmlToDocument` processor.

In another example, suppose that a parser uses an `AddString` transformer to add a prefix to the output of a `Content` anchor. The auto-generated serializer does not remove the prefix. If you need to remove it, you can insert a component such as a `Replace` transformer.

**Creating a Serializer by Editing the Script**

1. At the global level of the script, double-click the ellipsis (…) symbol, type a name for the serializer, and then press ENTER.
2. To the right of the equals sign, double-click the ellipsis, select a `Serializer`, and then press ENTER.
3. Expand the tree under the `Serializer` component. Assign its properties as required.
4. Add a schema that defines the XML syntax of the serializer input.
5. Under the `contains` line, add a sequence of nested serialization anchors and actions.
6. Run and test the serializer and modify the script as required.

**Online Sample**

For an example of a serializer that we created by editing the script, open the project `samples\Projects\ManualSerializer\ManualSerializer.cmw`. You can run the serializer on the input file `Example XML of Person.xml`.

**Creating a Serializer within a RunSerializer Action**

In addition to defining a serializer at the global level, it is possible to define a serializer within a `RunSerializer` action.

**Serialization Anchors**

Serialization anchors are the main components you use in a serializer. These are analogous to the anchors that are used in a parser, except that they work in the opposite direction. Anchors read data from locations in the source document and write the data to XML. Serialization anchors read XML data and write the data to locations in the output document.

The most important serialization anchors are `ContentSerializer` and `StringSerializer`.

- A `ContentSerializer` writes the content of a specified data holder to the output document. It is the inverse of a `Content` anchor, which reads content from a source document.
- A `StringSerializer` writes a predefined string to the output. It is the inverse of a `Marker` anchor, which finds a predefined string in a source document.
Example of Serialization Anchors

The following example illustrates three serialization anchors:

```
Psi_serialize = Serializer >>
  contains
  StringSerializer >>
    str = 'First Name'
  ContentSerializer >>
    opening_str = ''
    closing_str = ''
    data_holder = /Person/Name/First
  StringSerializer >>
    str = '+013-010 Last Name'
```

The first `StringSerializer` instructs the serializer to write the following text in the output document:

```
Firs
```

The `ContentSerializer` writes the value of the `Person/Name/First` element to the output.

The second `StringSerializer` writes the string:

```
< newline> Last Name: 
```

Note: The IntelliScript editor represents the newline and tab with ASCII codes and «, respectively.

Now, assume that you run the serializer on the following XML:

```
<Person gender="M">
  <Name>
    <First>Ron</First>
    <Last>Lehrer</Last>
  </Name>
  <Id>547329876</Id>
  <Age>27</Age>
</Person>
```

From the illustrated serialization anchors, the output is:

```
First Name: <tab>Ron
```

The display of this text is:

```
First Name: Ron
```

The serializer contains additional serialization anchors, which are not shown in the above illustration. The complete output of the serializer is:

```
First Name: Ron
Last Name: Lehrer
Id: 547329876
Age: 27
Gender: M
```

Sequence of Serialization Anchors

A serializer executes the serialization anchors in the sequence of their definitions.

Serialization anchors write data sequentially, always appending it to the end of the output document. You can alter the order by changing the sequence in the serializer configuration.

You can intersperse actions with the serialization anchors. The actions are executed as part of the sequence.
Standard Serializer Properties

The following table describes standard properties of the Serializer component and in many serialization anchors:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled   | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name       | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notifications | A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
| on_fail    | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional   | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark     | A user-defined comment that describes the purpose or action of the component. |

Serializer Component Reference

A Serializer converts XML documents to output documents in any format. It uses serialization anchors to identify and manipulate data in the source.

Serializer

A Serializer converts XML documents to output documents.
The following table describes the properties of the **Serializer** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_transformers</td>
<td>Defines a list of transformers that the <strong>Serializer</strong> applies to all serialized data.</td>
</tr>
</tbody>
</table>
| example_source      | Defines a sample XML source document. When you run the serializer in the Developer tool, it operates on the sample document. You can choose one of the following options:  
  - Empty. You are prompted for a source document when you run the serializer. Default.  
  - InputPort. Defines an input port.  
  - LocalFile. Defines a file on the local file system.  
  - Text. Defines a string.  
  - URL. Defines a URL.                                                                 |
| name                | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the name property to identify the component that caused the event. |
| notifications       | A list of **NotificationHandler** components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
| on_fail             | The action to take if the component fails. You can choose one of the following options:  
  - Cleared. Take no action.  
  - CustomLog. Write to the user log.  
  - LogError. Write an error message to the engine log.  
  - LogInfo. Write an information message to the engine log.  
  - LogWarning. Write a warning message to the engine log.  
  - NotifyFailure. Send a notification.  
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| output_file_extension| Defines the file extension of the generated output file. Default is ".txt".                                                                   |
| remark              | A user-defined comment that describes the purpose or action of the component.                                                                    |
| root_tag            | Defines the name of a root XML element that is not in the input schema for the serializer. **Note:** If the input of the serializer in XML from another component in the script, and the Data Processor transformation settings add a wrapper root element around the output, you must set this property to the name of the wrapper root element. |
| source              | Defines a data holder that contains the source for the serialization. For more information, see “Overview of Locators, Keys, and Indexing” on page 253. |
| target              | Defines a data holder that contains the result of the serialization. For more information, see “Overview of Locators, Keys, and Indexing” on page 253. |
| validate_source_document| Determines the level of source XML validation that the serializer performs. You can choose one of the following options:  
  - Partial. Permits some deviations from the schema. Default.  
  - Strict. Enforces the schema strictly.                                                                 |
The **AlternativeSerializers** serialization anchor defines a set of alternative serialization anchors that are nested below the parent serializer. Define a criterion for the alternative that the serializer should accept. Only the accepted alternative affects the serializer output. The other serialization anchors have no effect on the output.

The following table describes the properties of the **AlternativeSerializers** serialization anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| on_fail  | The action to take if the component fails. You can choose one of the following options:  
  - Cleared. Take no action.  
  - CustomLog. Write to the user log.  
  - LogError. Write an error message to the engine log.  
  - LogInfo. Write an information message to the engine log.  
  - LogWarning. Write a warning message to the engine log.  
  - NotifyFailure. Send a notification.  
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
  - Selected. Component failure does not cause the parent component to fail.  
  - Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
| selector | Defines the criterion for selecting one of the alternative serialization anchors. You can choose one of the following options:  
  - ScriptOrder. The serializer tests the nested serialization anchors in the order that they are defined in the script. It accepts the first one that succeeds. If all the nested serialization anchors fail, the **AlternativeSerializers** component fails.  
  - NameSwitch. The serializer searches for the nested serialization anchor whose **name** property is specified in a data holder. It ignores the other nested serialization anchors. If the named serialization anchor fails, the **AlternativeSerializers** component fails.  
  Default is ScriptOrder. |

**Example**

The input XML might contain a **Product** element or a **Service** element, but not both. You want to serialize whichever element is in the input.

Define an **AlternativeSerializers** serialization anchor and set its **selector** property to **ScriptOrder**.

Under the **AlternativeSerializers** component, nest two **ContentSerializer** serialization anchors. Configure one of them to process the **Product** element and the other to process **Service**.
**ContentSerializer**

The **ContentSerializer** serialization anchor writes the serialized data to the output document.

The following table describes the properties of the **ContentSerializer** serialization anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| allow_empty_values     | Determines whether **data_holder** can be empty. You can choose one of the following options:  
- **Selected.** The **data_holder** can be empty.  
- **Cleared.** The **data_holder** cannot be empty, otherwise the **ContentSerializer** fails.  
Default is cleared. |
| closing_str            | Defines the string that the anchor writes after the **data_holder**.                                                                                                                                                                                                                                                                   |
| data_holder            | Defines the data holder that contains the serialized data.                                                                                                                                                                                                                                                                            |
| disabled               | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- **Selected.** The script ignores the component.  
- **Cleared.** The script applies the component.  
Default is cleared. |
| ignore_default_transformers | Determines whether the default transformers of the **Serializer** are applied to the serialized data. You can choose one of the following options:  
- **Selected.** The default transformers of the **Serializer** are not applied.  
- **Cleared.** The default transformers of the **Serializer** are applied.  
Default is cleared. |
| name                   | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event.                                                                                                                     |
| on_fail                | The action to take if the component fails. You can choose one of the following options:  
- **Cleared.** Take no action.  
- **CustomLog.** Write to the user log.  
- **LogError.** Write an error message to the engine log.  
- **LogInfo.** Write an information message to the engine log.  
- **LogWarning.** Write a warning message to the engine log.  
- **NotifyFailure.** Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| opening_str            | Defines the string that the anchor writes before the contents of the **data_holder**.                                                                                                                                                                                       |
| optional               | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- **Selected.** Component failure does not cause the parent component to fail.  
- **Cleared.** Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark                 | A user-defined comment that describes the purpose or action of the component.                                                                                                                                                                                                                                    |
| transformers           | Defines a list of transformers that are applied to the serialized data.                                                                                                                                                                                                                                                             |
**DelimitedSectionsSerializer**

The **DelimitedSectionsSerializer** serialization anchor processes sections of data. The sections of the output are separated by a defined separator string.

Under the **DelimitedSectionsSerializer**, nest other serialization anchors. Each nested serialization anchor outputs a single section.

The following table describes the properties of the **DelimitedSectionsSerializer** serialization anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| **disabled** | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| **name** | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| **notifications** | A list of NotificationHandler components that handle notifications from nested components. For more information, see "Notifications" on page 307. |
| **on_fail** | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- NotifyFailure. Send a notification.  
The default is cleared. For more information about handling component failures, see "Failure Handling" on page 289. |
| **optional** | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
The default is cleared. For more information about component failure, see "Failure Handling" on page 289. |
| **remark** | A user-defined comment that describes the purpose or action of the component. |
| **separator** | Defines a serializer that defines the separator string. You can choose one of the following options:  
- AlternativeSerializers  
- ContentSerializer  
- EmbeddedSerializer  
- GroupSerializer  
- RepeatingGroupSerializer  
- StringSerializer  
- User-defined serializer  
The default is StringSerializer. |
Table

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>separator_position</td>
<td>Defines the position of the separator relative to the sections. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- after. Writes a separator after each section, including the first sections. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>- around. Writes separators before and after each section, including the first sections. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>- before. Write a separator before each section, including the first sections. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>- between. Writes a separator between the successive sections, but not before the first section and not after the last section. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Default is before.</td>
</tr>
<tr>
<td>using_placeholders</td>
<td>Determines whether the DelimitedSectionsSerializer writes the separator of an optional section that is missing from the XML input. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- always. Always writes the separator of a missing section. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>- never. Never writes the separator of a missing section. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>- when necessary. Always writes the separator of a missing internal section. Never writes the separator of a missing terminal section. For example,</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Default is when necessary.</td>
</tr>
</tbody>
</table>

Example

The XML input contains an employee resume. You want to write the data to an output text document in the following format:

```
----------------------------------
Jane Palmer
Employee ID 123456
----------------------------------
Professional Experience
...                          
----------------------------------
Education
...                          
```

Define a DelimitedSectionsSerializer with the line of hyphens as its separator. Because you want a line of hyphens before each section, set separator_position to before.

Within the DelimitedSectionsSerializer, nest three GroupSerializer components. The first GroupSerializer writes the Jane Palmer section, the second writes the Professional Experience section, and so forth.

Optional Sections

In this example, suppose that the second section, Professional Experience, is missing from some input XML documents, but you want to write its separator to the output, like this:

```
----------------------------------
Jane Palmer
Employee ID 123456
----------------------------------
Education
...                          
```
To support this situation, configure the **DelimitedSectionsSerializer** in the following way:

- In the second **GroupSerializer**, select the **optional** property. This means that if the **GroupSerializer** fails, it should not cause the **DelimitedSectionsSerializer** to fail.

- In the **DelimitedSectionsSerializer**, set **using_placeholders** to **true**. This means to write the separator of an optional section, even if the section itself is missing.

Alternatively, suppose that if the **Professional Experience** section is missing, you do not want to write its separator:

```
-----------------------------
Jane Palmer
Employee ID 123456
-----------------------------
Education
...
```

In this case, configure the **DelimitedSectionsSerializer** as follows:

- In the second **GroupSerializer**, select the **optional** property.
- In the **DelimitedSectionsSerializer**, set **using_placeholders** to **false**. This means not to write the separator of a missing section.

### EmbeddedSerializer

The **EmbeddedSerializer** serialization anchor activates a secondary **Serializer**, which writes its output in the same output document.

The following table describes the properties of the **EmbeddedSerializer** serialization anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
  The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| on_fail  | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
  Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
  Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
schema_connections
Connects the data holders that are referenced in the secondary serializer to the data holders that are referenced in the parent serializer. The property contains a list of Connect subcomponents that define the correspondences. For more information, see “Connect” on page 151.

If all the data holders in the main and secondary serializers are identical, you can omit this property. If there are any differences between the data holders, you must connect the data holders explicitly, even the ones that are identical.

serializer
Defines the name of a secondary serializer that is defined at the global level of the script.

Example

The XML input is a family tree. The input contains Person elements, which are recursively nested as shown:

```xml
<Person> <!-- Parent -->
    ...<Person> <!-- Child -->
    ...<Person> <!-- Grandchild -->
    ...
</Person>
</Person>
</Person>
```

In a Serializer, an EmbeddedSerializer component can call itself recursively until all levels of nesting are exhausted.

In this example, the schema_connections property connects Person to Person/Person. This instructs the secondary instance of the serializer to process a nested level of the input. When the two Person elements have the same data type, it is sufficient to connect just the parent element (Person), and not the nested elements (Person/*s/Name, Person/*s/BirthDate, etc.)

GroupSerializer

The GroupSerializer serialization anchor binds its nested serialization anchors together. You can set properties of the GroupSerializer that affect the members of the group.

The following table describes the properties of the GroupSerializer serialization anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>notifications</td>
<td>A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| on_fail  | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
| source   | Defines a data holder that contains the source for the serialization. For more information, see “Overview of Validators, Notifiers, and Failure Handling” on page 288. |
| target   | Defines a data holder that contains the result of the serialization. For more information, see “Overview of Validators, Notifiers, and Failure Handling” on page 288. |

**RepeatingGroupSerializer**

The **RepeatingGroupSerializer** serialization anchor writes a repetitive structure to the output document.

Use a **RepeatingGroupSerializer** when the XML data contains a multiple-occurrence data holder. It iterates over the occurrences of the data holder and outputs the data. For more information, see “Multiple-Occurrence Data Holders” on page 103.

Under the **RepeatingGroupSerializer**, nest serialization anchors that process and output each occurrence of the data holder. You can define a separator that the **RepeatingGroupSerializer** writes to the output between the iterations.

The following table describes the properties of the **RepeatingGroupSerializer** serialization anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Defines the number of iterations to run. If this property is blank, the iterations continue until the input is exhausted.</td>
</tr>
<tr>
<td>current_iteration</td>
<td>Defines a data holder where the <strong>RepeatingGroupSerializer</strong> outputs the number of the current iteration. You can use a <strong>ContentSerializer</strong> to write the number to the output.</td>
</tr>
</tbody>
</table>
| disabled   | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>notifications</td>
<td>A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307.</td>
</tr>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Take no action.</td>
</tr>
<tr>
<td></td>
<td>- CustomLog. Write to the user log.</td>
</tr>
<tr>
<td></td>
<td>- LogError. Write an error message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogInfo. Write an information message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogWarning. Write a warning message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- NotifyFailure. Send a notification.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>on_iteration_fail</td>
<td>Determines the action when an iteration fails. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Cleared. No action.</td>
</tr>
<tr>
<td></td>
<td>- CustomLog. Writes to the user log.</td>
</tr>
<tr>
<td></td>
<td>- LogError. Writes an error message to the Engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogInfo. Writes an information message to the Engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogWarning. Writes a warning message to the Engine log.</td>
</tr>
<tr>
<td></td>
<td>- NotifyFailure. Triggers a notification.</td>
</tr>
<tr>
<td></td>
<td>Use on_iteration_fail to write an entry when a single iteration fails. Use the on_fail property to write an entry when the entire RepeatingGroupSerializer fails. For more information, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>separator</td>
<td>Defines a serialization anchor that defines the separator string. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- AlternativeSerializers</td>
</tr>
<tr>
<td></td>
<td>- ContentSerializer</td>
</tr>
<tr>
<td></td>
<td>- EmbeddedSerializer</td>
</tr>
<tr>
<td></td>
<td>- GroupSerializer</td>
</tr>
<tr>
<td></td>
<td>- RepeatingGroupSerializer</td>
</tr>
<tr>
<td></td>
<td>- StringSerializer</td>
</tr>
<tr>
<td></td>
<td>- User-defined serializer</td>
</tr>
<tr>
<td></td>
<td>Default is blank.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>separator_position</td>
<td>Defines the position of the separator relative to the sections. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- after. Writes a separator after each section, including the first sections. For example,</td>
</tr>
<tr>
<td></td>
<td>1 2 3 4</td>
</tr>
<tr>
<td></td>
<td>- around. Writes separators before and after each section, including the first sections. For example,</td>
</tr>
<tr>
<td></td>
<td>1 2 3 4</td>
</tr>
<tr>
<td></td>
<td>- before. Write a separator before each section, including the first sections. For example,</td>
</tr>
<tr>
<td></td>
<td>1 2 3 4</td>
</tr>
<tr>
<td></td>
<td>- between. Writes a separator between the successive sections, but not before the first section and not after the last section. For example,</td>
</tr>
<tr>
<td></td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>skip_failed_iterations</td>
<td>Determines whether failed iterations are skipped. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. RepeatingGroup skips over a failed iteration and proceeds with the next iteration. If an iteration succeeds, the RepeatingGroup succeeds.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. RepeatingGroup fails if any iteration fails.</td>
</tr>
<tr>
<td></td>
<td>The skip_failed_iterations property has an effect only if separator is defined.                                                                <em>IDX11</em></td>
</tr>
<tr>
<td></td>
<td>Default is selected.</td>
</tr>
<tr>
<td>source</td>
<td>Defines a data holder that contains the source for the serialization. For more information, see “Overview of Validators, Notifiers, and Failure Handling” on page 288.</td>
</tr>
<tr>
<td>target</td>
<td>Defines a data holder that contains the result of the serialization. For more information, see “Overview of Validators, Notifiers, and Failure Handling” on page 288.</td>
</tr>
</tbody>
</table>

**Example**

The XML input contains the following structure:

```xml
<Persons>
  <Person>
    <Name>John</Name>
    <Age>35</Age>
  </Person>
  <Person>
    <Name>Larissa</Name>
    <Age>42</Age>
  </Person>
  ...
</Persons>
```

A **RepeatingGroupSerializer**, using a newline character as a separator, can output this data to:

```
John 35
Larissa 42
```

You can iterate over several multiple-occurrence data holders in parallel. For example, you can iterate over a list of men and a list of women, and output a list of married couples. To do this, insert a **ContentSerializer** within the repeating group for each data holder.

**StringSerializer**

The **StringSerializer** serialization anchor writes a predefined string to the output document.
The following table describes the properties of the **StringSerializer** serialization anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the <strong>name</strong> property to identify the component that caused the event.</td>
</tr>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options: - Cleared. Take no action. - CustomLog. Write to the user log. - LogError. Write an error message to the engine log. - LogInfo. Write an information message to the engine log. - LogWarning. Write a warning message to the engine log. - NotifyFailure. Send a notification. Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>str</td>
<td>Defines the string that the serializer writes to the output.</td>
</tr>
</tbody>
</table>
Chapter 15

Mappers

This chapter includes the following topics:

- Creating a Mapper, 243
- Components Nested within a Mapper, 243
- Mapper Example, 244
- Standard Mapper Properties, 245
- Mapper Component Reference, 246
- Mapper Anchor Component Reference, 247

Creating a Mapper

1. Add input and output schemas to the schema object, and then reference the schemas in the Data Processor transformation.
2. At the global level of the script, add a Mapper component.
3. Assign the source and target properties of the Mapper to the input and output elements of the Mapper, respectively.
4. Assign the example_source property to a sample XML input document.
   As you add components to the mapper, the Developer tool color-codes the corresponding locations in the example source. The colors can help you confirm that the components are defined correctly.
5. Edit the other properties of the Mapper as required.
6. Within the Mapper, nest a sequence of Map actions, mapper anchors, and any other required components.
7. Test the mapper and modify the script.

Components Nested within a Mapper

Within a Mapper, you can nest the following components:

- Any number of Map actions. The actions retrieve a data holder from the output and write the content to the output.
- Optionally, any number of mapper anchors. For more information, see the “Mapper Anchor Component Reference” on page 247.
• Optionally, any number of additional actions.

The Map actions and the mapper anchors can be in any sequence. You can also insert other actions in the sequence.

Notice that a mapper uses Map actions rather than mapper anchors to write to the output XML. This may seem a little different from parsers and serializers, where the output is created by anchors and serialization anchors, respectively. Actually, this is just a terminology issue. The Map action could have been defined as a mapper anchor. It is defined as an action because it is useful in other circumstances, unrelated to mappers.

Mapper Example

To illustrate the mapper configuration, we present a simple example.

Source XML

The input of the mapper is an XML document containing a list of personal names and their associated ID numbers.

```
<Persons>
  <Person ID="10">Bob</Person>
  <Person ID="17">Larissa</Person>
  <Person ID="13">Marie</Person>
</Persons>
```

Output XML

The desired output of the mapper is an XML list of the names and ID numbers, with no association between them.

```
<SummaryData>
  <Names>
    <Name>Bob</Name>
    <Name>Larissa</Name>
    <Name>Marie</Name>
  </Names>
  <IDs>
    <ID>10</ID>
    <ID>17</ID>
    <ID>13</ID>
  </IDs>
</SummaryData>
```
Mapper Configuration

The following mapper configuration performs the desired transformation:

```
<Mapper>
  source = /Persons

  target =
    <Locator>
      data_holder = /SummaryData
    </Locator>

  contains
    <RepeatingGroupMapping>
      contains
        <Map>
          source = /Persons/s/Person
          target = /SummaryData/s/Names/s/Name
          transformers = []
        </Map>
        <Map>
          source = /Persons/s/Person/@ID
          target = /SummaryData/s/IDs/s/ID
          transformers = []
        </Map>
    </RepeatingGroupMapping>
</Mapper>
```

The `RepeatingGroupMapping` iterates over the `Person` elements of the input. It uses `Map` actions to write the data to the `Name` and `ID` elements of the output.

Standard Mapper Properties

The following table describes standard properties in the `Mapper` component and in many mapper anchors:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled  | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name      | A descriptive label for the component. This label appears in the log file and the `Events` view. Use the `name` property to identify the component that caused the event. |
| notifications | A list of `NotificationHandler` components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
on_fail The action to take if the component fails. You can choose one of the following options:
- Cleared. Take no action.
- CustomLog. Write to the user log.
- LogError. Write an error message to the engine log.
- LogInfo. Write an information message to the engine log.
- LogWarning. Write a warning message to the engine log.
- NotifyFailure. Send a notification.
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.

optional Determines whether a component failure causes the parent component to fail. You can choose one of the following options:
- Selected. Component failure does not cause the parent component to fail.
- Cleared. Component failure causes the parent component to fail.
Default is cleared. For more information about component failure, see “Failure Handling” on page 289.

remark A user-defined comment that describes the purpose or action of the component.

### Mapper Component Reference

A mapper reads an XML source document and converts it to another XML document.

**Mapper**

A Mapper performs XML-to-XML transformations. It converts a source XML document to an output document that has a different XML structure.

The following table describes the properties of the Mapper component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| example_source | Defines a sample XML source document. When you run the mapper in the Developer tool, it operates on the sample document. You can choose one of the following options:  
- Empty. You are prompted for a source document when you run the script.  
- InputPort. Defines an input port.  
- LocalFile. Defines a file on the local filesystem.  
- Text. Defines a string.  
- URL. Defines a URL.  
Default is empty. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notifications | A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>on_fail</td>
<td>The action to take if the component fails. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Take no action.</td>
</tr>
<tr>
<td></td>
<td>- CustomLog. Write to the user log.</td>
</tr>
<tr>
<td></td>
<td>- LogError. Write an error message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogInfo. Write an information message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogWarning. Write a warning message to the engine log.</td>
</tr>
<tr>
<td></td>
<td>- NotifyFailure. Send a notification.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>root_tag</td>
<td>Defines the name of a root XML element that is not defined in the schema.</td>
</tr>
<tr>
<td></td>
<td>For example, if the top-level element of the schema is <code>Person</code>, but the XML input nests <code>Person</code> in an element called <code>InputWrapper</code>, set <code>root_tag</code> to <code>InputWrapper</code>.</td>
</tr>
<tr>
<td>source</td>
<td>Defines a locator component that defines an XML data holder. The data holder contains the root of the XML source for the mapping. For more information, see “Overview of Locators, Keys, and Indexing” on page 253.</td>
</tr>
<tr>
<td>target</td>
<td>Defines a locator component that defines an XML data holder. The data holder contains the root of the output XML for the mapping. For more information, see “Overview of Locators, Keys, and Indexing” on page 253.</td>
</tr>
<tr>
<td>validate_source_document</td>
<td>Determines the level of source XML validation that the serializer performs. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Partial. Permits some deviations from the schema.</td>
</tr>
<tr>
<td></td>
<td>- Strict. Enforces the schema strictly.</td>
</tr>
<tr>
<td></td>
<td>Default is Partial.</td>
</tr>
</tbody>
</table>

You must use the `source` and `target` properties to identify the root elements of the XML documents. For example, if the document element of the source is `Persons`, and the document element of the output is `SummaryData`, set the `source` and `target` as follows:

```
<Mapper
  source = Locator
data_holder = Persons
  target = Locator
data_holder = SummaryData
>`
AlternativeMappings

The **AlternativeMappings** mapper anchor defines a set of alternative mapper anchors. Define a criterion for selecting one alternative. Only the accepted alternative affects the output.

The following table describes the properties of the **AlternativeMappings** mapper anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| on_fail | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark | A user-defined comment that describes the purpose or action of the component. |
| selector | Determines the criterion for selecting one alternative mapper. You can choose one of the following options:  
- ScriptOrder. The script tests the nested mapper anchors in the sequence that they are defined in the script. It accepts the first one that succeeds. If all the nested mapper anchors fail, the **AlternativeMappings** component fails.  
- NameSwitch. The script searches for the nested mapper anchor whose name property is specified in a data holder. It ignores the other nested mapper anchors. If the named mapper anchor fails, the **AlternativeMappings** component fails.  
Default is ScriptOrder. |

**Example**

The input XML can contain a **Product** element or a **Service** element, but not both. You wish to process whichever element is in the input.

Define an **AlternativeMappings** mapper anchor, and set its selector property to **ScriptOrder**.

Within the **AlternativeMappings**, nest two **Map** actions. Configure one of them to process the **Product** element and the other to process **Service**.
The **EmbeddedMapper** mapper anchor activates a secondary **Mapper**, which stores its output in the same output document.

The following table describes the properties of the **EmbeddedMapper** mapper anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| **disabled**     | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared.                                                                                                                                                                                                 |
| **mapper**       | Defines the name of the secondary mapper.                                                                                                                                                                                                                                                                                                    |
| **name**         | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event.                                                                                                                                                           |
| **on_fail**      | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289.                                                                                                                                                  |
| **optional**     | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289.                                                                                                                                                  |
| **remark**       | A user-defined comment that describes the purpose or action of the component.                                                                                                                                                                                                                                                                   |
| **schema_connections** | Connects the data holders that are referenced in the secondary mapper to the data holders that are referenced in the parent mapper. The property contains a list of **Connect** subcomponents that define the correspondences. For more information, see “Connect” on page 151.  
If all the data holders in the main and secondary mappers are identical, you can omit this property. If there are any differences between the data holders, you must connect the data holders explicitly. |

**Example**

The XML input is a family tree. The input contains **Person** elements that are recursively nested as shown:

```xml
<Person>
  <!-- Parent -->
  ...
  <Person>  <!-- Child -->
    ...
    <Person>  <!-- Grandchild -->
      ...
    </Person>
  </Person>
</Person>
```
A Mapper can use an EmbeddedMapper component to call itself recursively until all levels of nesting are exhausted.

In this example, Person is connected to Person/Person. This instructs the secondary instance of the mapper to process a nested level of the input. When the two Person elements have the same data type, it is sufficient to connect just the parent element (Person), and not the nested elements (Person/*/s/Name, Person/*/s/BirthDate, etc.)

GroupMapping

The GroupMapping mapper anchor binds its nested mapper anchors and actions together. You can set properties of the GroupMapping that affect the members of the group.

The following table describes the properties of the GroupMapping mapper anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| absent   | Determines the behavior of GroupMapping when one of its mandatory, nested mapper anchors or actions fails. You can choose one of the following options:  
- Selected. GroupMapping succeeds.  
- Cleared. GroupMapping fails.  
Use this feature to test for the absence of nested mapper anchors. Default is cleared. |
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notifications | A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
| on_fail  | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
Default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark   | A user-defined comment that describes the purpose or action of the component. |
Property | Description
--- | ---
source | Defines a locator component that defines an XML data holder. The data holder contains the root of the XML source for the mapping. For more information, see “Overview of Locators, Keys, and Indexing” on page 253.
target | Defines a locator component that defines an XML data holder. The data holder contains the root of the output XML for the mapping. For more information, see “Overview of Locators, Keys, and Indexing” on page 253.

**RepeatingGroupMapping**

The **RepeatingGroupMapping** mapper anchor processes a repetitive structure in the input or output.

Use a **RepeatingGroupMapping** when the XML input or output contains a multiple-occurrence data holder. It iterates over occurrences of the data holders. For more information, see “Multiple-Occurrence Data Holders” on page 103.

Under the **RepeatingGroupMapping**, nest mapper anchors and actions that process each occurrence of the data holder.

The following table describes the properties of the **RepeatingGroupMapping** mapper anchor:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Defines the number of iterations to run. If this property is blank, the iterations continue until the input is exhausted.</td>
</tr>
<tr>
<td>current_iteration</td>
<td>Defines a data holder where the <strong>RepeatingGroupMapping</strong> outputs the number of the current iteration.</td>
</tr>
</tbody>
</table>
| disabled       | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name           | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notifications  | A list of NotificationHandler components that handle notifications from nested components. For more information, see “Notifications” on page 307. |
| on_fail        | The action to take if the component fails. You can choose one of the following options:  
- Cleared. Take no action.  
- CustomLog. Write to the user log.  
- LogError. Write an error message to the engine log.  
- LogInfo. Write an information message to the engine log.  
- LogWarning. Write a warning message to the engine log.  
- NotifyFailure. Send a notification.  
The default is cleared. For more information about handling component failures, see “Failure Handling” on page 289. |
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>on_iteration_fail</td>
<td>Determines the action when an iteration fails. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Cleared. No action.</td>
</tr>
<tr>
<td></td>
<td>- CustomLog. Writes to the user log.</td>
</tr>
<tr>
<td></td>
<td>- LogError. Writes an error message to the Engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogInfo. Writes an information message to the Engine log.</td>
</tr>
<tr>
<td></td>
<td>- LogWarning. Writes a warning message to the Engine log.</td>
</tr>
<tr>
<td></td>
<td>- NotifyFailure. Triggers a notification.</td>
</tr>
<tr>
<td></td>
<td>Use on_iteration_fail to write an entry when a single iteration fails. Use the on_fail property to write an entry when the entire RepeatingGroupMapping fails. For more information, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>skip_failed_iterations</td>
<td>Determines whether failed iterations are skipped. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. RepeatingGroup skips over a failed iteration and proceeds with the next iteration. If an iteration succeeds, the RepeatingGroup succeeds.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. RepeatingGroup fails if any iteration fails.</td>
</tr>
<tr>
<td></td>
<td>The skip_failed_iterations property has an effect only if separator is defined. Default is selected.</td>
</tr>
<tr>
<td>source</td>
<td>Defines a data holder that contains the source for the mapping. For more information, see “Overview of Locators, Keys, and Indexing” on page 253.</td>
</tr>
<tr>
<td>target</td>
<td>Defines a data holder that contains the result of the mapping. For more information, see “Overview of Locators, Keys, and Indexing” on page 253.</td>
</tr>
</tbody>
</table>

**Example**

For more information, including an example of a RepeatingGroupMapping, see the “Mapper Example” on page 244.
Overview of Locators, Keys, and Indexing

In designing a transformation, a frequent issue is how to locate the data holders that you want to process. If the same data holders can occur multiple times in an XML structure, there can be ambiguities in identifying the occurrences. This chapter explains how to use the Locator and Key components to resolve the ambiguities.

The components described in this chapter let you identify the occurrences of multiple-occurrence data holders in three ways:

- **Sequentially.** Each iteration of a component processes the next occurrence of the data holder.
- **By occurrence number.** For example, a component can select the third occurrence of a data holder.
- **By a key such as an attribute or a nested element.** The key uniquely identifies the occurrence of the data holder.

The sequential approach is the default. It is subject to some complexities that you can control by using the Locator component.

The occurrence number and key approaches are collectively known as indexing. The term is analogous to the index of a book, where you use a page number or a subject key to identify the location of information. You can implement the indexing by using components called `LocatorByOccurrence`, `LocatorByKey`, and `Key`.

You can use the locator and key components in parsers, serializers, or mappers. You can use the components to identify the occurrences of data holders in the input, the output, or both.

The locator components are nested in the source and target properties of various transformation components. The meaning and usage of the source and target properties is explained below.
Example of Locators

To understand the issues involved in identifying data holders, consider the following example. The example illustrates the use of:

- The target property
- The Locator component

We will explain the broad outline of the example here. In the following sections of the chapter, we will go back and explain how the target and the Locator work in detail.

Input and Output

Suppose that the output schema of a parser supports the following structure:

```xml
<Report>
  <Company>
    <Employee>John</Employee>
    <Employee>Leslie</Employee>
    <Employee>Pedro</Employee>
  </Company>
  <Company>
    <Employee>Marie</Employee>
    <Employee>Larry</Employee>
    <Employee>Frances</Employee>
  </Company>
</Report>
```

The source document that the parser processes is a list containing a single employee per company:

John
Marie

The output of the parser should be:

```xml
<Report>
  <Company>
    <Employee>John</Employee>
  </Company>
  <Company>
    <Employee>Marie</Employee>
  </Company>
</Report>
```

Incorrect Solution

Suppose you use the following `RepeatingGroup` to parse the source document:

```xml
<Report>
  <Company>
    <Employee>John</Employee>
  </Company>
  <Company>
    <Employee>Marie</Employee>
  </Company>
</Report>
```

The output is incorrect:
The problem is that both Company and Employee are multiple-occurrence elements. The RepeatingGroup creates multiple Employee elements correctly, but it does not know that each Employee element should be nested in a separate Company element.

**Correct Solution**

To resolve the ambiguity, you can assign the target property of the RepeatingGroup.

The target identifies the data holder that the RepeatingGroup should create. The target contains a Locator component pointing to the Company element. This means that each iteration of the RepeatingGroup should create a new occurrence of the Company element.

If you configure the RepeatingGroup in this way, the output is correct.

**Example of Indexing by Key**

To further introduce the data-holder identification issues, we present an example of indexing by key.

The example is a mapper that uses indexing to identify the occurrences of data holders in both its input and its output. On the input side, the indexing matches the corresponding data from different parts of an XML structure. On the output side, the indexing finds the correct location of an element in an XML structure.

The example illustrates the use of:

- The source and target properties
- The Locator, Key, and LocatorByKey components

In the following sections of the chapter, we will explain the detailed operation of these properties and components.

**Input**

The input XML is a report listing the names of parents and their children.

- For each parent, the XML lists a first name, a last name, and an ID.
For each child, the XML lists a first name, a hobby, and the ID of the parent.

```xml
<Report>
  <Parents>
    <Parent id="1" firstName="John" lastName="Smith"/>
    <Parent id="2" firstName="Jane" lastName="Doe"/>
  </Parents>
  <Children>
    <Child name="Eric" hobby="Swimming" parentID="1"/>
    <Child name="Elizabeth" hobby="Biking" parentID="2"/>
    <Child name="Mary" hobby="Painting" parentID="1"/>
    <Child name="Edward" hobby="Swimming" parentID="2"/>
  </Children>
</Report>
```

Output

The desired output is a list of hobbies and the children who engage in each hobby.

```xml
<Hobbies>
  <Hobby name="Swimming">
    <Person firstName="Eric" lastName="Smith"/>
    <Person firstName="Edward" lastName="Doe"/>
  </Hobby>
  <Hobby name="Biking">
    <Person firstName="Elizabeth" lastName="Doe"/>
  </Hobby>
  <Hobby name="Painting">
    <Person firstName="Mary" lastName="Smith"/>
  </Hobby>
</Hobbies>
```

Outline of the Transformation Approach

The transformation uses the following approach:

1. In the input XML, the transformation identifies the corresponding `Child` and `Parent` elements as follows:
   
   id attribute of Parent = parentID attribute of Child

2. The transformation creates `Hobby` and `Person` elements. It identifies the `Hobby` element where it should nest each `Person` element as follows:

   name attribute of Hobby = hobby attribute of Child

3. The transformation writes the child's first name into the `Person` element.

4. The transformation writes the parent's last name into the `Person` element.

Mapper Configuration

Key components define identifiers for the data holders.

- The first Key specifies that the id attribute is a unique identifier of a Parent element.
• The second Key specifies that the name attribute is a unique identifier of a Hobby element.

```xml
<ParentKey = Key >>
<recurring_element = /Report/*/Parents/*/Parent
<unique_fields =
   /Report/*/Parents/*/Parent/@id
>
```

```xml
<HobbyKey = Key >>
<recurring_element = /Hobbies/*/Hobby
<unique_fields =
   /Hobbies/*/Hobby/@name
>
```

The script then defines a Mapper with the following configuration:

```xml
<ReplicatingGroupMapping >>
<source =
   <Locater >>
      <data_holder = /Report
>
<target =
   <Locater >>
      <data_holder = /Hobbies
>
```

The components of the Mapper perform the following functions:

• The source property of the ReplicatingGroupMapping specifies that each iteration gets its input from the following data holders:
  - From an occurrence of the Child element
  - From the corresponding occurrence of the Parent element

• The target property of the ReplicatingGroupMapping specifies that each iteration stores its output in the following data holders:
  - In an occurrence of the Person element
  - In the corresponding occurrence of the Hobby element

• The first Map action copies the name attribute of the Child to the firstName attribute of the Person.

• The second Map action copies the lastName attribute of the Parent into the lastName attribute of the Person.
Use of Indexing

The example uses indexing by key to identify the occurrences of the Parent and Hobby data holders.

- In the source property of the RepeatingGroupMapping, the indexing identifies the occurrence of Parent that corresponds to a Child.
- In the target property, the indexing identifies the occurrence of Hobby where a Person should be nested.

Source and Target Properties

The source and target properties exist in components such as the following:

- In parsers:
  - Parser
  - Group
  - RepeatingGroup
  - EnclosedGroup
  - FindReplaceAnchor
- In serializers:
  - Serializer
  - GroupSerializer
  - RepeatingGroupSerializer
- In mappers:
  - Mapper
  - GroupMapping
  - RepeatingGroupMapping

In all these categories, the meaning and usage of the properties is identical:

- The source property identifies existing data holders that a transformation should use.
- The target property identifies data holders that may or may not already exist. If they exist, the transformation uses them. If they do not exist, the transformation creates them.

After you define the source and/or the target, the subsequent components use the identified data holders. For example, if you define the target of a Group, the anchors nested within the Group use the data holders that the target identifies.

Note: There are properties called source and target in some other components such as Map. These properties have a different meaning and usage from the above. For an explanation, please see the components where the properties are used.
Source Property

The **source** property identifies existing occurrences of data holders. The value of the **source** property is a list containing one or more of the following components:

<table>
<thead>
<tr>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locator</td>
<td>Identifies a single-occurrence or multiple-occurrence data holder. In the latter case, each iteration accesses the next occurrence, in sequence.</td>
</tr>
<tr>
<td>LocatorByKey</td>
<td>Identifies an occurrence of a multiple-occurrence data holder by using a key.</td>
</tr>
<tr>
<td>LocatorByOccurrence</td>
<td>Identifies an occurrence of a multiple-occurrence data holder by number.</td>
</tr>
</tbody>
</table>

**Default Behavior**

If you do not assign the **source** property of a component, the component identifies data holders in the following way:

- If there is only one occurrence of the data holder, the component uses the existing occurrence.
- If there are multiple occurrences of the data holder, the behavior is as follows:
  - In an iterative context, such as within a `RepeatingGroupSerializer`, each iteration accesses the next occurrence of the data holder in sequence.
  - In a non-iterative context, such as a `GroupSerializer` that is not nested within an iterative component, the component accesses the first occurrence of the data holder.

**Ambiguities in the Default Behavior**

There can be some ambiguities in the default behavior. Ambiguities can arise, for example, in the following circumstances.

- In cases where a multiple-occurrence element is nested within another multiple-occurrence element. For more information, see “Example 1: Nested Multiple-Occurrence Data Holders” on page 260.
- In cases where the schema permits alternative data holders, defined with `xs:choice`.
- In cases where the schema permits a data holder to be missing, defined with `minOccurs = 0`.

In such cases, it is prudent to assign the **source** property explicitly.

**Data Holder Must Exist**

The **source** property identifies a data holder that already exists in the scope of the transformation. If the data holder does not exist, the component containing the **source** property fails.

For example, suppose that the source property of a `Group` contains a non-optional `LocatorByOccurrence` that points to the third occurrence of a data holder. If only two occurrences exist, the `Group` fails.

**Using the Source Property for Input or Output**

Typically, a component uses the **source** property to identify where it should obtain input. For example, a `GroupSerializer` can use the property to identify an occurrence that it should serialize.

It is also possible to use the property to identify where the component should store output. For example, suppose that a parser has already created 10 occurrences of an XML element. After the occurrences have been created, a `GroupSerializer` could use the **source** property to identify the occurrences that it should serialize.
anchor assigns an attribute in one occurrence of the element. The `Group` can use the `source` property to identify the occurrence.

Example 1: Nested Multiple-Occurrence Data Holders

Suppose that the input schema of a serializer requires the following structure:

```xml
<Report>
  <Company>
    <Employee>John</Employee>
    <Employee>Leslie</Employee>
    <Employee>Pedro</Employee>
  </Company>
  <Company>
    <Employee>Marie</Employee>
    <Employee>Larry</Employee>
    <Employee>Frances</Employee>
  </Company>
</Report>
```

You want to iterate over all the `Employee` elements and produce the following output:

```
John
Leslie
Pedro
Marie
Larry
Frances
```

You might create a `RepeatingGroupSerializer` and configure it to output the `Employee` data holder.

```xml
data Holder = /Report/*/Company/*/Employee
```

This does not work correctly. By default, each iteration selects a new instance of `Employee` within the same `Company`. The result is the following output:

```
John
Leslie
Pedro
```

In other words, the `RepeatingGroupSerializer` accesses only the first `Company`. 
You can solve the problem by nesting the `RepeatingGroupSerializer` inside another `RepeatingGroupSerializer`. To resolve any potential ambiguities, you can configure the source properties explicitly.

```
|-- EmployeeSerializer
    |-- RepeatingGroupSerializer
        |-- separator_position = before
        |    |-- source =
        |    |    |-- Locator
        |    |    |    |-- data_holder = /Report/*/s/Company
        |    |    |    |-- ... contains
        |    |-- ... contains
        |    |-- RepeatingGroupSerializer
        |    |    |-- separator_position = before
        |    |    |    |-- source =
        |    |    |    |    |-- Locator
        |    |    |    |    |    |-- data_holder = /Report/*/s/Company/*/s/Employee
        |    |    |    |    |    |-- ... contains
        |    |    |-- ContentSerializer, /Report/*/s/Company/*/s/Employee
        |    |    |    |-- ... contains
        |    |    |-- ... contains
    |-- ... contains
```

Each iteration of the outer `RepeatingGroupSerializer` processes a different occurrence of `Company`. Each iteration of the nested `RepeatingGroupSerializer` processes a different occurrence of `Employee`. The result is the desired output.

Alternatively, suppose you want to iterate only over the second `Employee` element in each `Company`. The desired output is:

Leslie
Larry

You can do this by configuring a single `RepeatingGroupSerializer, whose source is Company`. This causes each iteration to access the next instance of `Company`. Within the iteration, you can configure a `GroupSerializer, whose source property uses a LocatorByOccurrence to select the second `Employee`. This generates the desired output.

```
|-- EmployeeSerializer
    |-- RepeatingGroupSerializer
        |-- separator_position = before
        |    |-- source =
        |    |    |-- Locator
        |    |    |    |-- data_holder = /Report/*/s/Company
        |    |    |    |-- ... contains
        |    |-- ... contains
        |    |-- GroupSerializer
        |    |    |-- source =
        |    |    |    |-- LocatorByOccurrence
        |    |    |    |    |-- recurring_element = /Report/*/s/Company/*/s/Employee
        |    |    |    |    |    |-- occurrence_number = 2
        |    |    |    |    |    |-- ... contains
        |    |    |-- ContentSerializer
        |    |    |    |-- opening_str = ""
        |    |    |    |    |-- closing_str = ""
        |    |    |    |    |-- data_holder = /Report/*/s/Company/*/s/Employee
        |    |    |    |-- ... contains
        |    |    |-- ... contains
    |-- ... contains
```
Example 2: Indexing

In the example of indexing by key at the beginning of this chapter, we used a `RepeatingGroupMapping` configured as shown below. In this example, the `source` property identifies two data holders:

- It uses a `Locator` component to identify an occurrence of `Child`. Each iteration processes the next occurrence of `Child`, sequentially.
- It uses a `LocatorByKey` component to identify an occurrence of `Parent`. This causes each iteration to process the occurrence of `Parent` that corresponds to the occurrence of `Child`.

```plaintext
< RepeatingGroupMapping >>
[ [ Locator >>
  [ data_holder = /Report/*/Children/*/Child
  [ LocatorByKey >>
    [ key = /Report/*/Parents/*/Parent[@id=51]
    [ params = /Report/*/Children/*/Child[@parentId]
  ]
  ]
]... ]
```

For more information, see “Example of Indexing by Key” on page 255.

Target Property

The `target` property identifies an occurrence of a data holder that may or may not already exist. If the occurrence exists, the component uses it. If the occurrence does not exist, the component creates it.

The value of the `target` property is a list containing one or more of the following components:

<table>
<thead>
<tr>
<th>Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locator</td>
<td>Identifies a single-occurrence or multiple-occurrence data holder. In the latter case, each iteration creates a new occurrence.</td>
</tr>
<tr>
<td>LocatorByKey</td>
<td>Identifies an occurrence of a multiple-occurrence data holder by an indexing key. If the occurrence does not yet exist, it is created.</td>
</tr>
<tr>
<td>LocatorByOccurrence</td>
<td>Identifies an occurrence of a multiple-occurrence data holder by number. If the occurrence does not yet exist, it is created along with any needed intervening occurrences. For example, if four occurrences exist, and <code>LocatorByOccurrence</code> specifies the tenth occurrence, occurrences 5-9 are also created, but left empty.</td>
</tr>
</tbody>
</table>

Default Behavior

If you do not assign the `target` property of a component, the component identifies data holders in the following way:

- If the schema permits only a single occurrence of the data holder, the script accesses or creates the occurrence.
- If the data holder can have multiple occurrences, the behavior is as follows:
  - In an iterative context, for example, within a `RepeatingGroup`, each iteration creates a new occurrence of the data holder.
  - In a non-iterative context, for example, a `Group` that is not nested within an iterative component, the component creates one new occurrence of the data holder.
Ambiguities in the Default Behavior

There can be some ambiguities in the default behavior. Ambiguities can arise, for example, in the following circumstances.

- In cases where a multiple-occurrence element is nested within another multiple-occurrence element. For more information, see “Example 1: Nested Multiple-Occurrence Data Holders” on page 263.
- In cases where the schema permits alternative data holders, defined with `xs:choice`.
- In cases where the schema permits a data holder to be missing, defined with `minOccurs = 0`.

In such cases, it is prudent to assign the `target` property explicitly.

Data Holder Can Be Created

The `target` property identifies a data holder that may or may not already exist in the scope of the transformation. If the data holder does not exist, it is created.

For example, suppose that the `target` property of a `Group` contains a `LocatorByKey`, which points to a particular occurrence of a data holder. If the occurrence already exists, the `Group` uses it. If the occurrence does not exist, the `Group` creates it.

Using the Target Property for Input or Output

Typically, a component uses the `target` property to identify where it should store output. For example, a `Group` can use the property to identify an occurrence where it should store data.

It is also possible to use the property to identify where a component should obtain input. For example, suppose that a `GroupSerializer` contains an action that computes data and stores it in a variable. The `GroupSerializer` then activates a `ContentSerializer` that writes the variable to the output. You can use the `target` property to create the occurrence of the variable that the `GroupSerializer` uses. The variable then serves as the input of the `ContentSerializer`.

Example 1: Nested Multiple-Occurrence Data Holders

The example of locators at the start of this chapter illustrates how to use the `target` property to differentiate between parent and child multiple-occurrence data holders. For more information, see “Example of Locators” on page 254.

Example 2: Indexing

The example of indexing by key at the start of this chapter, illustrates how to use the `target` property with indexing. The following figure illustrates how to configure the `target` property of the `RepeatingGroupMapping`:

```
<...>
  <Locators>
    <data_holder = /Hobbies/*/hobby/*Person
    <LocatorByKey
      key = /Hobbies/*/hobby[@name='S1']
      params =
        /Report/*/Children/*/Child/@hobby
    ...
  ...
```

The `target` property identifies the following data holders:

- A `Locator` component identifies an occurrence of `Person`. Each iteration creates a new occurrence of `Person`. 
A **LocatorByKey** component identifies the occurrence of the **Hobby** element, where the occurrence of **Person** should be nested. If the **Hobby** element already exists, the transformation uses it. If the **Hobby** element does not yet exist, the transformation creates it.

For more information, see “Example of Indexing by Key” on page 255.

**Standard Locator and Key Properties**

The following table describes standard properties that are used in the locator and key components:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

**Locator and Key Component Reference**

Locator and key components identify elements in the script or data holders.

**Key**

The **Key** component defines attributes or elements that serve as a unique identifier of their parent element.

You can define a **Key** only at the global level of the script, and you can reference the **Key** anywhere in the script. The name of a **Key** is case sensitive.

The following table describes the properties of the **Key** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled       | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| recurring_element | Defines a multiple-occurrence element whose occurrences are identified by the key. |
Optionally, you can define a list of data holders as a composite key. To do this, nest multiple data holders under the

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>unique_fields</td>
<td>Defines the key.</td>
</tr>
</tbody>
</table>

_example_

The **Example of Indexing by Key** defines a key for the Hobby element in the following structure:

```xml
<Hobbies>
  <Hobby name="Swimming">
    <Person firstName="Eric" lastName="Smith"/>
    <Person firstName="Edward" lastName="Doe"/>
  </Hobby>
  <Hobby name="Biking">
    <Person firstName="Elizabeth" lastName="Doe"/>
  </Hobby>
  <Hobby name="Painting">
    <Person firstName="Mary" lastName="Smith"/>
  </Hobby>
</Hobbies>
```

The key is the `name` attribute, which uniquely identifies each Hobby.

**Composite Keys**

Optionally, you can define a list of data holders as a composite key. To do this, nest multiple data holders under the `unique_fields` property.

Consider the following example:

```xml
<Persons>
  <Person ID="17" SubID="A">Bob</Person>
  <Person ID="17" SubID="B">Jane</Person>
  <Person ID="35" SubID="A">Larry</Person>
</Persons>
```

Neither the `ID` attribute nor the `SubID` attribute identifies a Person element uniquely. The combination of `ID` and `SubID`, however, is a unique identifier. You can define `ID` and `SubID` as a composite key.

**Restrictions on the Key**

The `unique_fields` must be nested within the `recurring_element`. They can be attributes of the element, they can be nested elements at any level of nesting, or they can be attributes of the nested elements.

For example, this means that `Persons/Person/SocialSecurity/@Number` can be a valid key for `Persons/Person`, because `@Number` is nested within `Persons/Person`. On the other hand, `Persons/Child` is not a valid key for `Persons/Person` because it is not correctly nested.

The `unique_fields` must identify the closest ancestor that can have multiple occurrences. For example, if both `Parent` and `Child` are multiple-occurrence elements, then `Parent/Child/@name` can be a valid key for `Parent/Child` but not for `Parent`. 

Locator and Key Component Reference 265
The unique fields must have simple data types. They cannot be structures.

**Sibling and Non-Sibling Occurrences**

A key uniquely identifies sibling occurrences of an element. It is permitted for non-sibling occurrences to have the same key.

Consider the following XML structure:

```xml
<Report>
  <Company>
    <Employee ID="1">John</Employee>
    <Employee ID="2">Leslie</Employee>
  </Company>
  <Company>
    <Employee ID="1">Marie</Employee>
    <Employee ID="2">Larry</Employee>
  </Company>
</Report>
```

The ID attribute can be a valid key for Employee because it uniquely identifies an Employee within a single Company. The duplication of ID values in different Company elements does not invalidate the key.

**Keys of Reusable Elements**

You can define a key on a reusable element that is defined in the schema.

For example, suppose that Persons/Person can occur in several different contexts within the XML. If you define ID as a key for Persons/Person, the key is valid in any context where Persons/Person is used.

**Enforced Uniqueness of a Key**

The script enforces the uniqueness of a Key. This has the following consequences:

- If two or more sibling occurrences of an input element have the same key values, the script considers each occurrence to overwrite the previous occurrences. It uses only the last occurrence that it encounters.
- If an occurrence of an input element is missing a key value, the occurrence is ignored.
- If the script outputs a keyed element, and a sibling element having the same key value already exists, the existing occurrence is overwritten.

In these cases, the script writes a warning in the event log.

**Locator**

The Locator component identifies a single-occurrence or a multiple-occurrence data holder in the source and target properties. For multiple-occurrence data holders, each iteration of a component that uses the Locator processes the next occurrence of the data holder.
The following table describes the properties of the **Locator** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_holder</td>
<td>Defines the data holder that the Locator component identifies.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

**LocatorByKey**

The **LocatorByKey** component identifies an occurrence of a multiple-occurrence data holder in the **source** and **target** properties.

Before you use **LocatorByKey**, you must define a **Key** at the global level of the script. The **Key** specifies the data holders that uniquely identify the occurrence.

The following table describes the properties of the **LocatorByKey** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>key</td>
<td>Defines the XPath predicate representation of the key. For example, if you have defined Hobbies/Hobby/@name as a Key, select Hobbies/Hobby[@name=$1]. This property is required.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>params</td>
<td>Defines the values of the parameters in the XPath predicate. Each value has a dollar sign ($) and an integer that represents the position of the parameter in the list of parameters. This property is required.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>
Conflicts Between Locators

Conflicts Between Locators

In case of conflicts, a nested **LocatorByKey** overrides a parent locator.

For example, suppose that the **target** property of a **Group** contains a **LocatorByKey** pointing to the third occurrence of an element. A nested **Group** contains a **LocatorByKey** pointing to the fifth occurrence. The nested **Group** uses the fifth occurrence.

LocatorByOccurrence

The **LocatorByOccurrence** component is used in the **source** property to identify an occurrence of a multiple-occurrence data holder.

For example:

- An element that can occur multiple times in an XML document
- A variable that can occur multiple times

The component identifies the occurrence by number. For example, if there are ten occurrences of a data holder, you can use **LocatorByOccurrence** to process the third occurrence. **LocatorByOccurrence** can be used to iterate over the occurrences in a repeating structure such as a **RepeatingGroup** anchor.

The **LocatorByKey** component identifies an occurrence of a multiple-occurrence data holder in the **source** and **target** properties.

Before you use **LocatorByKey**, you must define a **Key** at the global level of the script. The **Key** specifies the data holders that uniquely identify the occurrence.

The following table describes the properties of the **LocatorByOccurrence** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>occurrence_number</td>
<td>Defines the number of the occurrence.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>recurring_element</td>
<td>Defines the data holder that the component identifies.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

Conflicts Between Locators

Conflicts Between Locators

In case of conflicts, a nested **LocatorByOccurrence** overrides a parent locator.

For example, suppose that the **target** property of a **Group** contains a **LocatorByOccurrence** pointing to the third occurrence of an element. A nested **Group** contains a **LocatorByOccurrence** pointing to the fifth occurrence. The nested **Group** uses the fifth occurrence.
CHAPTER 17

Streamers

This chapter includes the following topics:
- Streamers Overview, 269
- Text Streamers, 270
- XML Streamers, 274
- Standard Streamer Properties, 276
- Streamer Component Reference, 276
- Streamer Subcomponent Reference, 284

Streamers Overview

A streamer splits a large source document into smaller portions that a transformation can process separately. Streamers are useful in transformations that process very large inputs, such as multi-gigabyte data streams.

A streamer offers the following advantages:
- The transformation parses each source segment as soon as it is available, rather than waiting until the entire source is received.
- The transformation has reduced memory requirements.

For example, suppose that an input stream contains stock market transaction data. The stream is transmitted to a server continuously over the course of the entire trading day. A streamer enables the script to process each transaction as soon as it arrives, rather than waiting until the end of the day.

In another example, suppose that you receive a large source file over an FTP connection. By using a streamer, the script can start processing the file before it is completely received.

The Data Processor transformation provides the following kinds of streamers:
- Streamer. Processes large text inputs.
- XmlStreamer. Process large XML inputs.

Streamers are runnable components. The Streamer or XmlStreamer component is defined at the global level of the script, and it must be set as the startup component of the transformation. It functions by splitting its input into segments and passing them to other runnable components, which can be parsers, mappers, or serializers.
Text Streamers

A Streamer component splits a large text document into smaller portions. The streamer divides the text source into header, footer, and repeating segments. As required by the source structure, the Streamer can subdivide the repeating segments into nested headers, footers, and repeating segments. The Streamer can pass each segment to an appropriate transformation.

Segments

A streamer identifies segments of its input. It passes the segments individually to transformations such as parsers, mappers, or serializers, which process the segment data.

A streamer assumes that the source is composed of:

- A header segment
- Any number of repeating segments
- A footer segment

For each type of segment, the streamer defines a transformation that processes the segment.

The repeating segments can be either simple or complex. A simple segment is a single unit of data. A complex segment has its own nested header, repeating segments, and footer.

Headers and footers are always simple segments.

Simple Segments

A simple segment has an opening marker that identifies where it starts, and a closing marker that identifies where it ends. Thus, a simple segment has the following structure:

```
Opening marker
Data
Closing marker
```

The streamer passes the segment to the specified transformation component, such as a parser.

It is possible to omit some of the markers from the streamer definition. For example:

- If you omit the opening marker of the source header, the header is assumed to start at the beginning of the source.
- If you omit the closing marker, then the segment ends at the opening marker of the next segment.

Complex Segments

A complex segment has a header and footer. Between the header and footer, it can contain any number of nested simple segments, for example:

```
Header
Simple segment
Simple segment
Simple segment
Footer
```

A complex segment can contain nested complex segments, for example:

```
Header
Complex segment
Complex segment
Complex segment
Footer
```
You can also define a complex segment that is missing the header or footer, for example:

Simple segment
Simple segment
Simple segment

The nested simple segments must all be of the same type. That is, they must all be identified by the same opening and closing markers.

Example

A data stream contains stock transaction data. The stream has the following structure:

- The header begins with the string yy-MM-dd/, which is a date followed by a slash.
- The header contains various data, followed by the string ENDHEAD/.
- The repeating segments begin with the string TRANS HH:mm nnn/, where HH:mm is the time on a 24-hour clock, and nnn is a serial number of any length.
- The data stream ends with the string END/.

The following is a sample data stream conforming to this specification, where ... represents arbitrary data that must be parsed:

06-12-13/...ENDHEAD/TRANS 09:30 1...TRANS 09:30 2...TRANS 09:31 03...TRANS 09:32 14...END/

You can parse this stream by using a streamer having the following schematic structure. Notice that the opening and closing markers are located by searching for a particular pattern or string.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Type</th>
<th>Opening Marker</th>
<th>Closing Marker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Simple</td>
<td>[0-9][0-9]-[0-9][0-9]-[0-9][0-9]-/</td>
<td>ENDHEAD/</td>
</tr>
<tr>
<td>Repeating</td>
<td>Simple</td>
<td>TRANS [0-9][0-9]-[0-9][0-9]-[0-9]+/</td>
<td>none</td>
</tr>
<tr>
<td>Footer</td>
<td>Simple</td>
<td>END/</td>
<td>none</td>
</tr>
</tbody>
</table>

Header Concatenation

Optionally, you can configure a streamer to concatenate the header segment with each of the repeating segments. The streamer passes the concatenated result to a transformation.

For example, suppose that a streamer passes the repeating segment to a parser. The source has the following structure, where Segment1 and so forth are instances of the repeating segment:

Header
Segment1
Segment2
Segment3

If you select the concatenation option, the streamer sends the following data to the parser:

HeaderSegment1
HeaderSegment2
HeaderSegment3

Output of a Streamer

A streamer generates an independent output document for each of the source segments.
Output in Design Environment

If you run a streamer in the design environment, it combines the individual output segments into a single output.

For example, suppose that the streamer passes each segment to a parser. The output of each parser is an XML document. The combined output is a sequence of XML documents, for example:

```xml
<xml version="1.0" encoding="windows-1252">
  <header>...</header>
</xml>

<xml version="1.0" encoding="windows-1252">
  <repeating_segment>...<repeating_segment>
</xml>
```

This output is not well-formed XML because it contains multiple root elements.

Wrapping Output in a Root Tag

You can wrap the combined output of a streamer in a root tag to convert the output to well-formed XML.

In the XML Generation tab of the Data Processor transformation settings, select Add XML root element and enter the name of the wrapper root element.

For example, if you specify a wrapper root element called MyRoot, the output becomes:

```xml
<MyRoot>
  <xml version="1.0" encoding="windows-1252">
    <header>...</header>
  </xml>

  <xml version="1.0" encoding="windows-1252">
    <repeating_segment>...<repeating_segment>
  </xml>
```

Using Markers and Variables in Streamers

Within a Streamer component, you cannot use the regular Marker and Variable components that are used in other types of transformations. Instead, you should use the MarkerStreamer component to define the opening and closing markers of simple segments. You can use the StreamerVariable component to store temporary data that is shared by all segments.

Creating a Streamer

1. Analyze the source structure and identify the segment types.
2. Create or open a script object.
3. In the script, configure a transformation such as a parser, mapper, or serializer that can process each type of simple segment.
4. In the same script, configure a Streamer component.
5. Within the Streamer, nest ComplexSegment and SimpleSegment components corresponding to the source structure.
6. For each **SimpleSegment**, define the opening marker and closing marker if required. Define the transformation that processes the segment.

7. Define the **Streamer** as the startup component.

**Streamer Configuration Example 1**

The following streamer contains simple segments. Each segment has a predefined opening and closing marker. The streamer passes the header and repeating segments to a parser called `body_p`. It passes the footer to a parser called `foot_p`.

**Streamer Configuration Example 2**

The following streamer contains a nested, repeating **ComplexSegment**. The nested **ComplexSegment** segment has its own header and nested, repeating **SimpleSegment**. The nested **ComplexSegment** does not have a footer.
Notice that the property `concat_header_to_repeating_segment` has been selected. The effect of this property is to concatenate the header to each instance of the repeating segment. The streamer passes the concatenated segments to the parser.

XML Streamers

An XMLStreamer component splits a large XML document into smaller portions. The XMLStreamer divides the XML source into header, footer, and body segments. The body segments can contain repeating or non-repeating elements. The XMLStreamer can pass each XML segment to an appropriate transformation, typically a mapper or a serializer.

An XMLStreamer works in much the same way as a Streamer, with a few differences due to the structured XML input. The following are the main features:

- The body segments are defined as XML elements. You can configure the body with multiple elements of the same or different types, in any sequence.
- The header is defined as the entire portion of the XML that precedes the first body segment. In the XMLStreamer configuration, it is not necessary to define the elements that comprise the header.
- The footer is defined as the entire portion of the XML that follows the last body segment. In the XMLStreamer configuration, it is not necessary to define the elements that comprise the footer.
- In many cases, the header and footer segments are not well-formed XML. To enable passing the segments to a mapper or serializer, you can configure modifier components that convert the segments to well-formed XML.

To help understand these features, consider the following source XML structure:

```xml
<stream>
  <headerline1>MainHeader</headerline1>
  <substreams>
    <substream>
      <subheaderline1>SubHeader</subheaderline1>
      <segments>
        <segment1>Segment1A</segment1>
        <segment1>Segment1B</segment1>
        <segment2>Segment2A</segment2>
      </segments>
    </substream>
  </substreams>
</stream>
```
In this example, you might define the body segments as the `substream` elements. The header is everything that precedes the first `substream`:

```
<stream>
  <headerline1>MainHeader</headerline1>
  <substreams>
    ...
  </substreams>
</stream>
```

The footer is everything that follows the last `substream`:

```
<stream>
  <footerline1>MainFooter</footerline1>
</stream>
```

The header and footer segments are not well-formed XML. You can apply modifiers that add closing or opening tags to make them well-formed. For example, a modifier can convert the header to:

```
<stream>
  <headerline1>MainHeader</headerline1>
  <substreams>
    ...
  </substreams>
</stream>
```

You can configure the `XmlStream`er to pass the header segment, the footer segment, and each instance of the `substream` segment to an appropriate transformation, such as a mapper or serializer.

Alternatively, you might subdivide the `substream` elements into `segment1` and `segment2` segments, and send each of these to its own mapper or serializer. Notice that `segment1` and `segment2` follow each other in a random sequence. The `XmlStream`er ignores the sequence and processes `segment1` and `segment2` in whatever order they occur.

The following figure illustrates the configuration for this purpose. The script defines independent serializers for the header, footer, `segment1`, and `segment2` segments.
As a further refinement, you can define transformations for the nested headers and footers within each substream element.

### Standard Streamer Properties

The following table describes common properties of streamer components:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
</tbody>
</table>

### Streamer Component Reference

Streamers split a large source document into smaller portions that a transformation can process separately.
ComplexSegment

A **ComplexSegment** component defines a source structure that contains a header, a repeating portion, and a footer.

The following table describes the properties of the **ComplexSegment** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| concat_header_to_repeatingSegment | Determines whether the system includes the **header_segment** with each instance of the **repeating_segment**. It passes the result to the **run_component** of the **repeating_segment**. You can choose one of the following options:  
  - Selected. Each repeating segment has a copy of the header.  
  - Cleared. Each repeating segment appears without the header.  
  For more information, see "Header Concatenation" on page 271. Default is cleared. |
| footer_segment                  | Defines the footer portion of the source. Under this property, you can nest a **SimpleSegment** that defines the footer. If this property is undefined, the script processes the source as if it has no footer. |
| header_segment                  | Defines the header portion of the source. Under this property, you can nest a **SimpleSegment** that defines the header. If this property is undefined, the script processes the source as if it has no header. |
| repeating_segment               | Defines the repeating portion of the source. Under this property, you can nest a **SimpleSegment** that defines the repeating data. You can also nest a **ComplexSegment** that has its own header-repeating-footer structure. |

ComplexXmlSegment

A **ComplexXmlSegment** component defines a nested structure within the body portion of an **XmlStreamer** input. The nested structure can have its own header, body, and footer.

Under a **ComplexXmlSegment**, you can nest **XmlSegment**, **ComplexXmlSegment**, and **SimpleXmlSegment** components.

The following table describes the properties of the **ComplexXmlSegment** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| allow_unmarked_text | Determines whether the segments in the **sub_elements** list can be separated by intervening text or by other elements. You can choose one of the following options:  
  - Selected. The segments can be separated by intervening text or by other elements. The intervening content is ignored.  
  - Cleared. The segments can be separated only by whitespace.  
  Default is cleared. |
| disabled           | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| Footer             | Defines how to process the footer of the **ComplexXmlSegment**. For more information, see "XmlSegment" on page 283. Default is XmlSegment. |
**Property** | **Description**
--- | ---
Header | Defines how to process the header of the **ComplexXmlSegment**. For more information, see "XmlSegment" on page 283. Default is XmlSegment.
locator | Defines a data holder.
name | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event.
remark | A user-defined comment that describes the purpose or action of the component.
sub_elements | Defines a list of **ComplexXmlSegment** or **SimpleXmlSegment** components that define how to process the body of the **ComplexXmlSegment**.

**MarkerStreamer**

A **MarkerStreamer** component defines the opening and closing markers of simple segments. It is similar to a regular **Marker** anchor, but it is used only in streamers.

The following table describes the properties of the **MarkerStreamer** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| adjacent | Determines whether the **MarkerStreamer** must be adjacent to the end of the preceding segment. You can choose one of the following options:  
- Selected. Requires **MarkerStreamer** to be adjacent to the end of the preceding segment.  
- Cleared. **MarkerStreamer** can be separated from the end of the preceding segment.  
Default is cleared. Use this property to ensure that the segments are not separated by any other text or whitespace. |
| count | Determines which occurrence of the marker to begin processing with. For example, set **count** to 3 to skip the first and second occurrences of the marker.  
This property is being phased out. It is available for compatibility with existing projects. Do not use it in new projects. |
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| marking | Determines whether the marker is used as a reference point to identify the next segment or marker. You can choose one of the following options:  
- begin position. Before only.  
- end position. After only.  
- full. Places a reference point before and after the current marker.  
Default is full. |
| name | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
Using the Marking Property to Define Segment Boundaries

You can use the marking property to control whether the data in the opening and closing marker is included in the segment and passed to a transformation.

The rule is that the streamer passes the data between the innermost reference points surrounding the segment. For example:

- If the opening marker has `marker = begin position`, the innermost reference point is at the start. The entire marker is included in the segment.
- If the opening marker has `marker = end position` or `full`, the innermost reference point is at the end. The marker is excluded from the segment.

The inverse relationships apply to the closing marker.

To illustrate this, consider a simple segment having the following structure:

```
BEGIN...data...END
```

A MarkerStreamer identifies the opening marker by searching for the text `BEGIN`. Another MarkerStreamer identifies the closing marker by searching for `END`.

The following table illustrates how the marking property affects the segment boundaries.

<table>
<thead>
<tr>
<th>Marking of Opening Marker</th>
<th>Marking of Closing Marker</th>
<th>Segment Passed to the Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>full</td>
<td>...data...</td>
</tr>
<tr>
<td>full</td>
<td>begin</td>
<td>...data...</td>
</tr>
<tr>
<td>full</td>
<td>end</td>
<td>...data...END</td>
</tr>
<tr>
<td>begin</td>
<td>full</td>
<td>BEGIN...data...</td>
</tr>
<tr>
<td>begin</td>
<td>begin</td>
<td>BEGIN...data...</td>
</tr>
<tr>
<td>begin</td>
<td>end</td>
<td>BEGIN...data...END</td>
</tr>
<tr>
<td>end</td>
<td>full</td>
<td>...data...</td>
</tr>
<tr>
<td>end</td>
<td>begin</td>
<td>...data...</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
<td>...data...END</td>
</tr>
</tbody>
</table>
SimpleSegment

A **SimpleSegment** component defines a data unit that contains an opening marker and a closing marker. It also defines the transformation that processes the data unit.

The opening and closing markers are defined with regular expressions. For more information about regular expression syntax, see “RegularExpression” on page 179.

The following table describes the properties of the **SimpleSegment** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>closing_marker</td>
<td>Defines a regular expression that identifies the segment end. If this property is undefined, the segment end is the end of the source or the start of the next segment. Default is MarkerStreamer.</td>
</tr>
<tr>
<td>count</td>
<td>Defines the maximum number of segments to pass to the transformation. For example, if <strong>count</strong> is 3, the streamer searches for three consecutive instances of the segment. It passes the three segments together to the transformation. If it finds only one or two segments, it passes those segments. If the segments are small, passing multiple segments to a transformation can improve performance because it reduces the streamer overhead. Within the transformation, use a component such as <strong>RepeatingGroup</strong> to process the individual segments. Default is 1.</td>
</tr>
</tbody>
</table>
| disabled       | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:
- Selected. The script ignores the component.
- Cleared. The script applies the component. The default is cleared. |
| name           | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| opening_marker | Defines a regular expression that identifies the segment start. If this property is undefined, the segment start is the beginning of the source or the end of the preceding segment. Default is MarkerStreamer. |
| remark         | A user-defined comment that describes the purpose or action of the component.                                                                                                                                 |
| run_component  | Defines a transformation that processes the segment. You can choose one of the following options:
- The name of a parser, serializer, or mapper that is configured at the global level of the script.
- A **Mapper** or **Serializer** component. Configure the mapper or serializer within the segment.
- A **WriteSegment** component that copies the segment to the output. For more information, see “WriteSegment” on page 287. |

SimpleXmlSegment

A **SimpleXmlSegment** component defines a body segment of an **XmlStreamer** input. It defines the element containing the segment and the transformation that should process the segment.

Because a **SimpleXmlSegment** is an XML element, the segment is always well-formed. You can apply a modifier that alters the segment before you pass the segment to a transformation.
The following table describes the properties of the SimpleXmlSegment component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Defines the maximum number of segments to pass to the transformation. For example, if count is 3, the streamer searches for three consecutive instances of the segment. It passes the three segments together to the transformation. If it finds only one or two segments, it passes those segments. If the segments are small, passing multiple segments to a transformation can improve performance because it reduces the streamer overhead. Within the transformation, use a component such as RepeatingGroup to process the individual segments. Default is 1.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options: - Selected. The script ignores the component. - Cleared. The script applies the component. The default is cleared.</td>
</tr>
<tr>
<td>locator</td>
<td>Defines a data holder.</td>
</tr>
<tr>
<td>modifier</td>
<td>Defines how the segment is modified before it is passed to a transformation. You can select the following modifier components: - AddHeaderModifier. Passes the segment together with the header of the XML section in which the segment is located. - AddStringModifier. Concatenates the segment with prefix or suffix strings. - DoNothingModifier. Does not modify the segment. - WellFormedModifier. Adds closing tags and/or a root element to ensure that the segment is well-formed XML. For more information about the modifiers, see the “Streamer Subcomponent Reference” on page 284. Default is DoNothingModifier.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>run_component</td>
<td>Defines a transformation that processes the segment. You can choose one of the following options: - The name of a parser, serializer, or mapper that is configured at the global level of the script. - A Mapper or Serializer component. Configure the mapper or serializer within the segment. - A WriteSegment component that copies the segment to the output. For more information, see “WriteSegment” on page 287.</td>
</tr>
</tbody>
</table>

**Streamer**

The **Streamer** component splits text input into segments. It passes each type of segment to a predefined transformation such as a parser, mapper, or serializer.

The **Streamer** must be defined at the global level of the script and it must be the startup component of the transformation.

Under a **Streamer**, you must nest a **ComplexSegment**. The **ComplexSegment** can contain nested **SimpleSegment** or **ComplexSegment** components.
The following table describes the properties of the **Streamer** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td>Defines the overall structure of the source. Default is ComplexSegment.</td>
</tr>
</tbody>
</table>
| disabled     | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| max_lookup_size | Defines the maximum quantity of new data, in kilobytes, that the Streamer searches for each new segment.  
  For optimal performance, set this property to twice the maximum possible segment size.  
  When an application activates a deployed Streamer service through an API, it must set the chunk size parameter to a value that is smaller than the max_lookup_size. Default is 10000. |
| name         | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| on_end_of_input | Defines a transformation that runs at the end of the input stream. For example, the transformation can output a summary message. You can choose one of the following options:  
  - The name of a parser, serializer, or mapper that is configured at the global level of the script.  
  - A Mapper or Serializer component. Configure the mapper or serializer within the streamer. |
| remark       | A user-defined comment that describes the purpose or action of the component. |
| root_tag     | Defines an XML tag in which the Streamer wraps the combined output from all the segments. For more information, see “Output of a Streamer” on page 271. |

**StreamerVariable**

A **StreamerVariable** component is a user-defined variable whose scope includes all segments of a **Streamer** or XmlStreamer.

For example, if a streamer contains three parsers, the value of a **StreamerVariable** is available to all three parsers. A parser that processes a header segment might retrieve data from the header and store it in the **StreamerVariable**. The other parsers, which process the repeating segment and the footer segment, can access the value of the **StreamerVariable**. You cannot use a regular **Variable** for this purpose because the value of the variable is not shared between segments.

In other respects, the **StreamerVariable** component is similar to a regular **Variable**. However, a **StreamerVariable** must have a simple, single-occurrence data type. For more information, see “Variables” on page 99.

You can define a **StreamerVariable** only at the global level of the script.
The following table describes the properties of the **StreamerVariable** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>An initial value for the <strong>StreamerVariable</strong>, assigned when the transformation starts. Select <strong>InitialValue</strong> and enter the value.</td>
</tr>
<tr>
<td>val_type</td>
<td>Defines the data type that the variable can store. Assign a simple type such as <code>xs:string</code> or <code>xs:integer</code>. Streamer variables cannot have complex or multiple-occurrence types. Default is <code>xs:string</code>.</td>
</tr>
</tbody>
</table>

**XmlSegment**

An **XmlSegment** component defines a header or footer segment of an **XmlStreamer** input. It also defines the transformation that processes the header or footer.

An unmodified header or footer is not necessarily well-formed XML. By assigning a modifier component, you can configure the **XmlSegment** to always return well-formed XML.

The following table describes the properties of the **XmlSegment** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| modifier | Defines how the segment is modified before it is passed to a transformation. You can select the following modifier components:  
- AddHeaderModifier. Passes the segment together with the header of the XML section in which the segment is located.  
- AddStringModifier. Concatenates the segment with prefix or suffix strings.  
- DoNothingModifier. Does not modify the segment. This is the default.  
- WellFormedModifier. Adds closing tags and/or a root element to ensure that the segment is well-formed XML.  
For more information about the modifiers, see the “Streamer Subcomponent Reference” on page 284. Default is WellFormedModifier. |
| name | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| remark | A user-defined comment that describes the purpose or action of the component. |
| run_component | Defines a transformation that processes the segment. You can choose one of the following options:  
- The name of a parser, serializer, or mapper that is configured at the global level of the script.  
- A **Mapper** or **Serializer** component. Configure the mapper or serializer within the segment.  
- A **WriteSegment** component that copies the segment to the output. For more information, see “**WriteSegment**” on page 287. |

**XmlStreamer**

The **XmlStreamer** component splits an XML input into header, body, and footer segments. It passes each type of segment to a predefined transformation such as a mapper or serializer.
The `XmlStreamer` must be defined at the global level of the script and it must be the startup component of the transformation.

Under an `XmlStreamer`, you can nest `XmlSegment`, `ComplexXmlSegment`, and `SimpleXmlSegment` components.

The following table describes the properties of the `XmlStreamer` component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| allow_unmarked_text    | Determines whether the segments in the `sub_elements` list can be separated by intervening text or by other elements. You can choose one of the following options:  
  - Selected. The segments can be separated by intervening text or by other elements. The intervening content is ignored.  
  - Cleared. The segments can be separated only by whitespace.  
  Default is cleared. |
| disabled               | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
  - Selected. The script ignores the component.  
  - Cleared. The script applies the component.  
  The default is cleared. |
| Footer                 | Defines how to process the footer of the `ComplexXmlSegment`. For more information, see “XmlSegment” on page 283. Default is `XmlSegment`. |
| Header                 | Defines how to process the header of the `ComplexXmlSegment`. For more information, see “XmlSegment” on page 283. Default is `XmlSegment`. |
| max_lookup_size        | Defines the maximum quantity of new data, in kilobytes, that the `XmlStreamer` searches for each new segment.  
  For optimal performance, set this property to twice the maximum possible segment size.  
  When an application activates a deployed `XmlStreamer` service through an API, it must set the chunk size parameter to a value that is smaller than the `max_lookup_size`.  
  Default is 10000. |
| name                   | A descriptive label for the component. This label appears in the log file and the `Events` view. Use the `name` property to identify the component that caused the event. |
| remark                 | A user-defined comment that describes the purpose or action of the component. |
| sub_elements           | Defines a list of `ComplexXmlSegment` or `SimpleXmlSegment` components that define how to process the body of the XML input. |

**Streamer Subcomponent Reference**

Streamer subcomponents modify segments of a `Streamer` or an `XmlStreamer`.

**AddHeaderModifier**

In an `XmlStreamer`, the `AddHeaderModifier` component adds the header of the current segment to the segment. The component adds closing tags as required to ensure that the result is well-formed XML.
You can use **AddHeaderModifier** to pass a segment to a transformation, in the context of its header.

The following table describes the properties of the **AddHeaderModifier** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name     | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| remark   | A user-defined comment that describes the purpose or action of the component. |

In the following example, `<segment1>` is a repeating segment, preceded by a header and followed by a footer.

```xml
<stream>  
  <headerline>...</headerline>  
  <segments>  
    <segment1>...</segment1>  
    <segment1>...</segment1>  
    <segment1>...</segment1>  
  </segments>  
  <footerline>...</footerline>  
</stream>
```

You can configure an **XmlStreamer** that returns the following header, which is not well-formed XML:

```xml
<stream>  
  <headerline>...</headerline>  
  <segments>
</stream>
```

If you apply **AddHeaderModifier** to `<segment1>`, the modifier prefixes each instance of `<segment1>` with the header. It adds closing tags to ensure that the XML is well-formed. The result is the following segment:

```xml
<stream>  
  <headerline>...</headerline>  
  <segments>  
    <segment1>...</segment1>  
  </segments>  
</stream>
```

If you apply **AddHeaderModifier** to a header segment, the modifier adds the header of the parent element to the segment. Do not apply **AddHeaderModifier** to the initial header of the **XmlStreamer** input, because the initial header does not itself have a parent element.

**AddStringModifier**

In an **XmlStreamer**, the **AddStringModifier** component adds strings before and after a segment.
The following table describes the properties of the **AddStringModifier** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled   | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name       | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| pre        | Defines the string before the segment.                                     |
| post       | Defines the string after the segment.                                      |
| remark     | A user-defined comment that describes the purpose or action of the component. |

**DoNothingModifier**

In an **XmlStreamer**, this component is a placeholder. It does not modify the segment to which it is applied.

**WellFormedModifier**

In an **XmlStreamer**, the **WellFormedModifier** component ensures that a segment is well-formed XML. It can add opening, closing, or root tags as required for this purpose.

The following table describes the properties of the **WellFormedModifier** component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled         | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name             | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| new_root_element | Defines the root element. The root element must be an ancestor of the segment. If you do not assign this property, the modifier does not add a root element. |
| remark           | A user-defined comment that describes the purpose or action of the component. |

Consider the following XML input:

```xml
<stream>
  <headerline>...</headerline>
  <substreams>
    <substream>
      <headerline>...</headerline>
      <segments>
        <segment1>...</segment1>
```
The Notice that the modifier added the closing tags. The result is the following well-formed segment:

```
<stream>
  <substream>
    <subheaderline>...</subheaderline>
  </substream>
</stream>
```

If you apply `WellFormedModifier` to this header, the modifier adds the closing tags. The result is the following well-formed segment:

```
<stream>
  <substream>
    <subheaderline>...</subheaderline>
  </substream>
</stream>
```

Now suppose that you configure `WellFormedModifier` to add `<stream>` as a root element. The result is:

```
<stream>
  <substreams>
    <substream>
      <subheaderline>...</subheaderline>
    </substream>
  </substreams>
</stream>
```

Notice that the modifier added the `<stream>` and `<substreams>` elements, preserving the skeleton structure of the original XML.

**WriteSegment**

The `WriteSegment` component copies a segment to a specified output location. The component does not alter the copied segment. The `WriteSegment` component is an option of the `run_component` property of the `XmlSegment` component.

The following table describes the properties of the `WriteSegment` component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>Defines the output location. The <code>output</code> property has the following options:</td>
</tr>
<tr>
<td></td>
<td>- <code>OutputDataHolder</code>. Writes to a data holder.</td>
</tr>
<tr>
<td></td>
<td>- <code>OutputFile</code>. Writes to a file.</td>
</tr>
<tr>
<td></td>
<td>- <code>OutputPort</code>. Defines the name of an <code>AdditionalOutputPort</code> where the data is written.</td>
</tr>
<tr>
<td></td>
<td>- <code>ResultFile</code>. Writes to the default results file of the transformation.</td>
</tr>
<tr>
<td></td>
<td>- <code>StandardErrorLog</code>. Writes to the user log. For more information, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td></td>
<td>For more information about these options, see “Action Subcomponent Reference” on page 224. Default is <code>ResultFile</code>.</td>
</tr>
</tbody>
</table>
CHAPTER 18

Validators, Notifications, and Failure Handling

This chapter includes the following topics:

- Overview of Validators, Notifiers, and Failure Handling, 288
- Failure Handling, 289
- Validators, 292
- Standard Validator Properties, 292
- Validator Component Reference, 293
- Notifications, 307
- Notification Component Reference, 308

Overview of Validators, Notifiers, and Failure Handling

When you design a transformation, you must consider the following questions:

- What happens if the input data is invalid? For example, a date might have the wrong format, a string might be too long, or the records might be out of sequence.
- What happens if data is missing from the input? For example, an address might omit the house number.
- What happens if the input has an unusual structure? For example, the records might be out of sequence.

Any of these conditions might occur because of an input error. If so, they can cause transformations errors and failures.

The conditions might also occur under normal circumstances. For example, an input protocol might permit certain fields to be missing.

You can incorporate transformation features that detect such conditions and take appropriate actions. The following approaches are among the possible actions:

- Fail the transformation and generate no output.
- Fail a portion of the transformation, roll back its output, but permit the transformation to generate output for other portions of the data.
- Continue the entire transformation, but write a message to a user log.
- Continue the entire transformation, but write a message to the result file of the transformation.
This chapter explains what happens in the event of a transformation failure, and how you can handle failure conditions. It then explains how you can detect data validation errors that might cause failures, and how you can write notifications about such conditions to the output.

Failure Handling

A failure is an event that prevents a component from processing data in the expected way. An anchor might fail if it searches for text that does not exist in the source document. A transformer or action might fail if its input is empty or has an inappropriate data type.

A failure can be a perfectly normal occurrence. For example, a source document might contain an optional date. A parser contains a Content anchor that processes the date, if it exists. If the date does not exist in a particular source document, the Content anchor fails.

By configuring the transformation appropriately, you can control the result of a failure. In the above example, you might configure the parser to ignore the missing data and continue processing.

The event log displays warnings about failures. In addition, you can configure a transformation to write a failure message in a user log.

Using the Optional Property to Handle Failures

You can use the optional property to control the behavior of a transformation when a failure occurs.

Failure Causes Parent to Fail

If the optional property of a component is not selected, a failure of the component causes its parent to fail. If the parent is also non-optional, its own parent fails, and so forth.

For example, suppose that a Parser contains a Group, and the Group contains a Marker. All the components are non-optional. If the Marker does not exist in the source document, the Marker fails. This causes the Group to fail, which in turn causes the Parser to fail.

Pictorially, we can represent these relationships in the following way:

```
Parser    //Failed
 Group     //Failed
  Marker   //Failed
```

Optional Failure Does Not Cause Parent to Fail

If the optional property of a component is selected, a failure of the component does not bubble up to the parent.

In the above example, suppose that the Group is optional. The failed Marker causes the Group to fail, but the Parser does not fail.

```
Parser    //Succeeded
 Group     //Failed
  Marker   //Failed
```

Rollback

If a component fails, its effects are rolled back.
For example, suppose that a Group contains three non-optional Content anchors that store values in data holders. If the third Content anchor fails, the Group fails. The script rolls back the effects of the first two Content anchors. The data that the first two Content anchors already stored in data holders is removed.

The rollback applies only to the main effects of a transformation, such as a parser storing values in data holders or a serializer writing to its output. The rollback does not apply to side effects. In the above example, if the Group contains a WriteValue action that writes a line in a text output file, the line is not deleted.

Setting the Optional Property
You can set the optional property of a component in the following ways:

- Edit the advanced properties of a component in the script.
- Right-click the component, and then click Make Optional or Make Mandatory.

Components that Lack an Optional Property
Certain components lack the optional property because the components never fail, regardless of their input.

An example is the Sort action. If the Sort action finds no data to sort, it simply does nothing. It does not report a failure.

Writing a Failure Message to the User Log
You can configure a component to output failure events to a user-defined log. For example, if an anchor fails to find text in the source document, it can write a message in the user log. This can occur even if the anchor is defined as optional, so that the failure does not terminate the transformation processing.

The user log can contain information such as:

- Failure level: Information, Warning, or Error
- Name of the component that failed
- Failure description
- Location of the failed component in the script
- Additional information about the transformation status, such as the values of data holders.

Configuring User Log Output
To define the user log output, assign the on_fail property of the appropriate transformation components. The following components have an on_fail property:

- Parsers and anchors
- Serializers and serialization anchors
- Mappers and mapper anchors

The on_fail property can have the following values:

- LogError. Writes an error message containing the VarLastFailure system variable to the user log.
- LogWarning. Same as LogError, but displays the message as a warning rather than an error.
- LogInfo. Same as LogError, but displays the message as information rather than an error.
- CustomLog. Runs a serializer that writes a custom message to the user log or another location. For more information, see “CustomLog” on page 203.
- NotifyFailure. Triggers a notification.
The following example illustrates a Marker anchor with a LogInfo configuration:

```
<Marker >>
  optional
  search = TextSearch >>
  text = "Height"
  on_fail = LogInfo
```

If the Marker does not exist in the source document, the system writes the following entry in the user log:

```
*** INFO *** : Marker, [MyParser[11].Marker], Can't find Marker<optional>'Height'.
```

**Viewing the User Log**

The user log is an ASCII text file. On Windows platforms, the default location of the user log is:

```
c:\Informatica\DataTransformation\UserLogs
```

On UNIX platforms, the default location is:

```
<INSTALL_DIR>/UserLogs
```

By default, each execution of a transformation generates a user log having a unique name:

```
<service_name>+<unique_string>.log
```

A transformation can set the user-log location at runtime by using SetValue actions to assign the following system variables. Set the phase property of SetValue to initial. Ensure that SetValue runs before any component that writes to the user log.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarServiceInfo &gt; StandardError &gt; StandardErrorDir</td>
<td>Directory path of the user log.</td>
</tr>
<tr>
<td>VarServiceInfo &gt; StandardError &gt; StandardErrorName</td>
<td>File name of the user log.</td>
</tr>
</tbody>
</table>

In the following example, a SetValue action sets the user-log directory to c:\mydirectory.

```
<SetValue >>
  disabled
  optional
  phase = initial
  quote = "c:\mydirectory"
  data_holder = $VarServiceInfo/*/s/StandardError/*/s/StandardErrorDir
  name = ""
  remark = ""
  on_fail = ...
  transformers = []
```
Validators

A validator component confirms that its input conforms to a condition. You can use validators to check input for maximum or minimum string lengths or numeric values, conformance with expressions, or many other conditions. You can apply multiple validators to the same input.

If the input does not conform to the condition, the validator triggers a notification. A NotificationHandler component can process the notification. For example, if you use validators in a parser, a NotificationHandler can insert a warning message in the parser output. For more information, see “Notifications” on page 307.

You can insert validators in locations such as the validators property of a Content anchor or Map action. The validators enable you to warn if the input is invalid, without necessarily failing the Content or Map.

In addition to the validators described in this chapter, you can validate data against a set of user-defined rules and generate an XML validation report. For more information, see “ValidateValue” on page 220.

Standard Validator Properties

The following table describes standard properties of validators:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| allow_empty_value| Determines whether an empty input is accepted as valid. You can choose one of the following options:  
- Selected. Empty input is valid.  
- Cleared. Empty input is not valid.  
Default is cleared. |
| disabled         | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name             | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| negation         | Determines whether the validation condition is negated. You can choose one of the following options:  
- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.  
- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.  
Default is cleared. |
| notify           | Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared. |
| optional         | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
Validator Component Reference

Validator components test input data for conformity to defined rules.

AlternativeValidators

The AlternativeValidators validator contains a set of nested validators that apply to the input. Use an AlternativeValidators to apply OR logic to a set of validation conditions. The data is valid if it satisfies any of the conditions.

The following table describes the properties of the AlternativeValidators validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_value</td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>negation</td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notify</td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark | A user-defined comment that describes the purpose or action of the component. |
| selector | Determines the criterion for selecting a validator from among the validators nested below the AlternativeValidators component. You can choose one of the following options:  
- ScriptOrder. The parser tests the nested validators in the sequence defined in the script. It accepts the first validator that succeeds. If all the validators fail, the input is invalid.  
- NameSwitch. The parser searches for the nested validator whose name property is specified in the data holder defined in option_name. It ignores the other validators. If the named validator fails, the input is invalid.  
Default is ScriptOrder. |
| transformers | Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered. |

**EDIFACTValidation**

The EDIFACTValidation validator tests whether a source string is a valid EDIFACT message.

The following table describes the properties of the EDIFACTValidation validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| enabled | Determines the setting for param1. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| param1 | Determines whether the input is optional. param1 is named is optional and has only one property, enabled. enabled has the following options:  
- Selected. The input data is optional.  
- Cleared. The input data is mandatory. |
<p>| param2 | Defines an EDI data type. param2 is named input_type and has only one property, value. value is a hard-keyed string or a data holder. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>param3</td>
<td>Defines a range of integers. param3 is named minmax_limits and has only one property, value. value is a hard-keyed string or a data holder that specifies two integers separated by a hyphen.</td>
</tr>
<tr>
<td>param4</td>
<td>Defines a list of values. param4 is named enumerations and has only one property, value. value is a hard-keyed string or a data holder that specifies a comma-separated list of strings or integers.</td>
</tr>
</tbody>
</table>

**Remark**: A user-defined comment that describes the purpose or action of the component.

**Value**: Defines a value for param1, param2, or param3.

**Note**: This component is deprecated. The IntelliScript editor displays it for legacy scripts. Do not use it in new scripts. Instead, use other validator components.

## Enumeration

The **Enumeration** validator tests whether a value is a member of a set of values.

The following table describes the properties of the **Enumeration** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| allow_empty_value | Determines whether an empty input is accepted as valid. You can choose one of the following options:  
- Selected. Empty input is valid.  
- Cleared. Empty input is not valid.  
Default is cleared. |
| disabled         | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| enumerations     | Defines a list of values. |
| ignore_case      | Determines whether the comparison is case sensitive. You can choose one of the following options:  
- Selected. The comparison is not case sensitive.  
- Cleared. The comparison is case sensitive.  
Default is cleared. |
| name             | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| negation         | Determines whether the validation condition is negated. You can choose one of the following options:  
- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.  
- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.  
Default is cleared. |
| notify           | Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared. |
**Property** | **Description**
--- | ---
optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:
- Selected. Component failure does not cause the parent component to fail.
- Cleared. Component failure causes the parent component to fail.
Default is cleared. For more information about component failure, see “Failure Handling” on page 289.

**remark** | A user-defined comment that describes the purpose or action of the component.

**transformers** | Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.

---

**LengthEquals**

The **LengthEquals** validator tests whether the length of a string is equal to a specified value.

The following table describes the properties of the **LengthEquals** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| allow_empty_value | Determines whether an empty input is accepted as valid. You can choose one of the following options:
- Selected. Empty input is valid.
- Cleared. Empty input is not valid.
Default is cleared. |
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:
- Selected. The script ignores the component.
- Cleared. The script applies the component.
The default is cleared. |
| length | Defines the length of the string. |
| name | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event. |
| negation | Determines whether the validation condition is negated. You can choose one of the following options:
- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.
- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.
Default is cleared. |
| notify | Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared. |
| optional | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:
- Selected. Component failure does not cause the parent component to fail.
- Cleared. Component failure causes the parent component to fail.
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
MaxLength

The **MaxLength** validator tests whether the length of a string is less than or equal to a specified value.

The following table describes the properties of the **MaxLength** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| allow_empty_value | Determines whether an empty input is accepted as valid. You can choose one of the following options:  
- Selected. Empty input is valid.  
- Cleared. Empty input is not valid.  
Default is cleared. |
| disabled        | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| length          | Defines the maximum length of the string.                                                                                                                                                                                                                                                                                                  |
| name            | A descriptive label for the component. This label appears in the log file and the **Events** view. Use the **name** property to identify the component that caused the event.                                                                                                                   |
| negation        | Determines whether the validation condition is negated. You can choose one of the following options:  
- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.  
- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.  
Default is cleared. |
| notify          | Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.                                                                                                           |
| optional        | Determines whether a component failure causes the parent component to fail. You can choose one of the following options:  
- Selected. Component failure does not cause the parent component to fail.  
- Cleared. Component failure causes the parent component to fail.  
Default is cleared. For more information about component failure, see “Failure Handling” on page 289. |
| remark          | A user-defined comment that describes the purpose or action of the component.                                                                                                                                                                                                   |
| transformers    | Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered. |

### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
</tbody>
</table>

**validators**

**MaxLength**
MaxNumber

The MaxNumber validator tests whether a number is less than or equal to a specified value.

The following table describes the properties of the MaxNumber validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_value</td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script.</td>
</tr>
<tr>
<td></td>
<td>You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component.</td>
</tr>
<tr>
<td>negation</td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notify</td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
<tr>
<td>value</td>
<td>Defines the maximum value of the number.</td>
</tr>
</tbody>
</table>

MinLength

The MinLength validator tests whether the length of a string is greater than or equal to a specified value.
The following table describes the properties of the **MinLength** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_value</td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>length</td>
<td>Defines the minimum length of the string.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>negation</td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notify</td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
</tbody>
</table>

**MinNumber**

The **MinNumber** validator tests whether a number is greater than or equal to a specified value.
The following table describes the properties of the **MinNumber** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_value</td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script.</td>
</tr>
<tr>
<td></td>
<td>You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>negation</td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notify</td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
<tr>
<td>value</td>
<td>Defines the minimum value of the number.</td>
</tr>
</tbody>
</table>

**NumberEquals**

The **NumberEquals** validator tests whether a number is equal to a specified value.
The following table describes the properties of the **NumberEquals** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_value</td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the <strong>name</strong> property to identify the component that caused the event.</td>
</tr>
<tr>
<td>negation</td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notify</td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
<tr>
<td>value</td>
<td>Defines the value of the number.</td>
</tr>
</tbody>
</table>

**ValidateByExpression**

The **ValidateByExpression** validator evaluates a JavaScript expression. If the expression is false, the validator considers the input to be invalid.
The following table describes the properties of the **ValidateByExpression** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_value</td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid. Default is cleared.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>expression</td>
<td>Defines a JavaScript expression. Use $0 for the validator input. Use a dollar sign ($) plus an integer for additional data holders defined under params, starting with $1. For example, the following expression checks whether the input has the value Ron Lehrer:</td>
</tr>
<tr>
<td></td>
<td>$0 == &quot;Ron Lehrer&quot;</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>negation</td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid. Default is cleared.</td>
</tr>
<tr>
<td>notify</td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>params</td>
<td>Defines a list of data holders that contain parameters for use in the expression.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
</tbody>
</table>

**ValidateByPattern**

The **ValidateByPattern** validator tests whether a string matches a regular expression. For more information, see “RegularExpression” on page 179.
The following table describes the properties of the **ValidateByPattern** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_value</td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. Use one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>expression</td>
<td>Defines a regular expression.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>negation</td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notify</td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
</tbody>
</table>

**ValidateByTransformer**

The **ValidateByTransformer** validator applies a list of one or more transformers to the input. If the list of transformers fails, the validator considers the input to be invalid.
The following table describes the properties of the ValidateByTransformer validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_value</td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script.</td>
</tr>
<tr>
<td></td>
<td>You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event.</td>
</tr>
<tr>
<td>negation</td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notify</td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>run_transformers</td>
<td>Defines a list of transformers.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
</tbody>
</table>

**ValidateByType**

The ValidateByType validator tests whether its input conforms to a specified data type.
The following table describes the properties of the **ValidateByType** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_value</td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the <strong>name</strong> property to identify the component that caused the event.</td>
</tr>
<tr>
<td>negation</td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notify</td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
<tr>
<td>val_type</td>
<td>Defines a data type. Select a standard type or a type that is defined in the project schemas.</td>
</tr>
</tbody>
</table>

**ValidateDate**

The **ValidateDate** validator tests whether a date conforms to a specified ICU date format, for example, `yyyy-MM-dd`. For more information, see “DateFormatICU” on page 164.
The following table describes the properties of the **ValidateDate** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_empty_value</td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>disabled</td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td>format_string</td>
<td>Defines an ICU date format.</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the <strong>name</strong> property to identify the component that caused the event.</td>
</tr>
<tr>
<td>negation</td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td>notify</td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td>optional</td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td>remark</td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td>transformers</td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
</tbody>
</table>

**ValidatorPipeline**

The **ValidatorPipeline** validator applies a list of validators to the data. If any of the validators reports invalidity, or if a validator is marked as **optional** and fails, the **ValidatorPipeline** triggers a notification.

Use a **ValidatorPipeline** to apply AND logic to a set of validation conditions. The data is valid if it satisfies all the conditions.
The following table describes the properties of the **ValidatorPipeline** validator:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>allow_empty_value</strong></td>
<td>Determines whether an empty input is accepted as valid. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Empty input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Empty input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td><strong>disabled</strong></td>
<td>Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. The script ignores the component.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. The script applies the component.</td>
</tr>
<tr>
<td></td>
<td>The default is cleared.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>A descriptive label for the component. This label appears in the log file and the <strong>Events</strong> view. Use the <strong>name</strong> property to identify the component that caused the event.</td>
</tr>
<tr>
<td><strong>negation</strong></td>
<td>Determines whether the validation condition is negated. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. If the condition is true, the input is not valid, and if the condition is false, the input is valid.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. If the condition is true, the input is valid, and if the condition is false, the input is not valid.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared.</td>
</tr>
<tr>
<td><strong>notify</strong></td>
<td>Defines the name of a notification. If the input does not conform to the validation condition, the validator triggers the notification. For more information, see “Notifications” on page 307. Default is cleared.</td>
</tr>
<tr>
<td><strong>optional</strong></td>
<td>Determines whether a component failure causes the parent component to fail. You can choose one of the following options:</td>
</tr>
<tr>
<td></td>
<td>- Selected. Component failure does not cause the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>- Cleared. Component failure causes the parent component to fail.</td>
</tr>
<tr>
<td></td>
<td>Default is cleared. For more information about component failure, see “Failure Handling” on page 289.</td>
</tr>
<tr>
<td><strong>remark</strong></td>
<td>A user-defined comment that describes the purpose or action of the component.</td>
</tr>
<tr>
<td><strong>transformers</strong></td>
<td>Defines a list of transformers that apply to the input. The validation condition is applied to the result of the transformers. The transformers have only a temporary effect on the data for the purpose of validation. The input is not permanently altered.</td>
</tr>
</tbody>
</table>

**Notifications**

A notification is a signal that a condition has occurred in a transformation. When the condition occurs, a transformation triggers the notification. You can configure handlers that process the notifications.

The following examples illustrate some ways to use notifications:

- A validator can trigger a notification. A **NotificationHandler** component can write a validation warning message to the result file of the transformation or to a log.
- A **StructureDefinition** anchor can define a set of **NotificationHandler** components to process mismatches between the input records and the required input structure. If a mismatch occurs, the appropriate **NotificationHandler** writes a message to the result file or to a log.
• A **Notify** action to trigger a notification in any location of a transformation. A **NotificationHandler** can write a message to the result file or to a log.

The following table describes the types of notifications:

<table>
<thead>
<tr>
<th>Notification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MandatoryStructureMissing</td>
<td>A mandatory record does not appear in the input.</td>
</tr>
<tr>
<td>MismatchIDs</td>
<td>The record and subelement IDs partially match. For example, there are two record identifiers, and only one of them matches.</td>
</tr>
<tr>
<td>StructureBelowMinOccurs</td>
<td>There are fewer matching records of the subelement than defined in <strong>minOccurs</strong>.</td>
</tr>
<tr>
<td>StructureExceedsMaxOccurs</td>
<td>There are more matching records of the subelement than defined in <strong>maxOccurs</strong>.</td>
</tr>
<tr>
<td>StructureOutOfSequence</td>
<td>The records match the subelements but not in the required sequence. For example, the subelements define a sequence <strong>ABC</strong>, but the input contains <strong>ACB</strong>.</td>
</tr>
<tr>
<td>UnexpectedRecord</td>
<td>The records match the subelements, but not in the required hierarchy. For example, the subelement define a sequence <strong>ABC</strong>, and <strong>D</strong> is defined in another location. The input contains <strong>ABD</strong>.</td>
</tr>
<tr>
<td>UnrecognizedRecord</td>
<td>No subelement matches any of the record identifiers.</td>
</tr>
<tr>
<td>XsdValidationError</td>
<td>The input does not match the requirements of the schema.</td>
</tr>
</tbody>
</table>

**Notification Component Reference**

Notification components perform actions when a component fails.

**Notification**

The component defines the name of a notification. Configure the component at the global level of the script.

The following figure shows a notification called **StringTooLong**:

`StringTooLong = Notification()`

**NotificationGroup**

The component defines a single name that refers to a set of notification names. Configure the component at the global level of the script.
The following example shows a group called StringToLongOrShort:

```java
StringTooLong = Notification()
StringTooShort = Notification()
StringTooLongOrShort = NotificationGroup
  notifications =
    StringTooLong
    StringTooLong
    ... 
```

You might configure a NotificationHandler to process StringTooLongOrShort. If a transformation triggers a StringTooLong or StringTooShort notification, the handler processes the notification.

The following table describes the properties of the NotificationGroup component.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>notifications</td>
<td>Defines a list of notifications.</td>
</tr>
</tbody>
</table>

**NotificationComponent**

The NotificationHandler component defines a list of actions to take for a specified notification.

Insert the NotificationHandler component in locations such as the notifications property of a Group or RepeatingGroup. Within the component, you can insert a WriteValue action that stores a message in a data holder.

The VarNotificationDetails variable stores information about the notification that was most recently triggered. A NotificationHandler writes the information stored in VarNotificationDetails to the output. For more information about VarNotificationDetails, see “System Variables” on page 99.

The following table describes the properties of the NotificationHandler component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| disabled | Determines whether the script ignores the component and all of the child components. Use this property to test, debug, and modify a script. You can choose one of the following options:  
- Selected. The script ignores the component.  
- Cleared. The script applies the component.  
The default is cleared. |
| name | A descriptive label for the component. This label appears in the log file and the Events view. Use the name property to identify the component that caused the event. |
| notification | Defines the name of a notification for NotificationHandler to process. Select a predefined notification or a notification name that is defined in a Notification or NotificationGroup component. To configure a handler that processes any notification, select anyNotification. |
| remark | A user-defined comment that describes the purpose or action of the component. |
| source | Defines a data holder that you can use for input to the NotificationHandler. |
| target | Defines a data holder for the output of the NotificationHandler. |
The following figure shows a Group anchor that is configured with a NotificationHandler:

If a component in the Group triggers an InvalidValueInGroup notification, the handler processes it. The handler writes the VarNotificationDetails/Value variable, together with a text string, to the result file of the transformation.

**NotifyFailure**

NotifyFailure is a possible value of the on_fail property of anchors and other components. If the component fails, NotifyFailure triggers a notification.

The following table describes the properties of the NotifyFailure component.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>notify</td>
<td>Defines the notification to trigger. Select a predefined notification or a notification name that is defined in a Notification or NotificationGroup component.</td>
</tr>
<tr>
<td>value</td>
<td>Defines the value of the VarNotificationDetails/Value variable. A NotificationHandler can include the value in its output.</td>
</tr>
</tbody>
</table>
Custom Script Components

This chapter includes the following topics:

- Custom Script Components Overview, 311
- Custom Component Example, 311
- Custom Component Properties, 312
- Developing a Custom Component in Java, 312
- Developing a Custom Component in C or C++, 314
- Configuring a Custom Component, 316

Custom Script Components Overview

When you design and configure a Data Processor transformation script, you can use a large number of built-in components. You can also program custom components, such as document processors or transformers, and insert them into a script. When you export the Data Processor transformation as a Data Transformation service, the service runs the custom components.

You can implement the custom components in Java, C, or C++.

For more information about the interfaces that you must implement, see the External-Component Java Interface Reference and the External-Component C and C++ Interface Reference.

Custom Component Example

Suppose you need to parse a proprietary binary data format. Rather than parse the binary data directly, you prefer to convert the data to a text representation that is easier to parse.
To do this, you can program a custom document processor, which you might call MyBinaryToText. The processor might have properties such as the following:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeepLineBreaks</td>
<td>A Boolean property. When true, the processor preserves the line-break characters in the binary data.</td>
</tr>
<tr>
<td>MaxLineLength</td>
<td>An integer property. Specifies the maximum length of the text lines to output.</td>
</tr>
<tr>
<td>Ignore</td>
<td>A string property. Tells the processor to ignore data fields beginning with the specified string.</td>
</tr>
</tbody>
</table>

After you develop the processor, you can install it and use it in scripts.

Custom Component Properties

The properties of a custom component can have integer, Boolean, string, or list-of-string data types. You can assign either a constant property value or the name of a data holder that contains the value.

You can hide some of the properties in the IntelliScript editor. For example, a custom component might support four properties. In its TGP configuration file, you can configure it to display only the first two properties. The script passes only the displayed properties to the component. The component can assign its own default values to the hidden properties.

The maximum number of properties depends on the component type and the language of implementation.

<table>
<thead>
<tr>
<th>Component type</th>
<th>Language</th>
<th>Maximum Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document processor</td>
<td>Java</td>
<td>4</td>
</tr>
<tr>
<td>Document processor</td>
<td>C or C++</td>
<td>5</td>
</tr>
<tr>
<td>Transformer</td>
<td>Java</td>
<td>10</td>
</tr>
<tr>
<td>Transformer</td>
<td>C or C++</td>
<td>10</td>
</tr>
</tbody>
</table>

Developing a Custom Component in Java

1. Create a class that implements one or more of the following interfaces:

<table>
<thead>
<tr>
<th>Component Type</th>
<th>Type of Input</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document processor</td>
<td>File</td>
<td>CMXFileProcessor</td>
</tr>
<tr>
<td>Document processor</td>
<td>Buffer</td>
<td>CMXByteArrayProcessor</td>
</tr>
</tbody>
</table>
Component Type | Type of Input | Interface
--- | --- | ---
Transformer | String | CMXStringTransformer
Transformer | Buffer | CMXByteArrayTransform

For more information about these interfaces, see the *External-Component Java Interface Reference*.

2. Compile the project to a JAR file.

3. Store the JAR in the `externLibs\user` subdirectory of the installation directory of every computer where you plan to use the component.

4. Create a script file that defines the display name of the component and its properties. Store the file in the `autoInclude\user` subdirectory of the installation directory.

   For more information about this step, see “Configuring a Custom Component” on page 316.

You can then use the custom component in transformations.

### Java Interface Example

As an example, consider a document processor that accepts file input. The processor must implement the CMXFileProcessor class, which has the following method:

```java
public String process(
    CMXContext context,
    String in,
    String additionalFilesDir,
    CMXEventReporter reporter)
    throws Exception
```

The meaning of the parameters is as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context</td>
<td>Input parameter. An object containing the properties that the script passes to the component. The <code>parameters</code> method of the object returns a vector containing the property values.</td>
</tr>
<tr>
<td>in</td>
<td>Input parameter. The full path of the file that the component operates on.</td>
</tr>
<tr>
<td>additionalFilesDir</td>
<td>Optional output parameter. The path of a temporary directory where the component writes files. At the end of processing, the script deletes the entire directory content.</td>
</tr>
<tr>
<td>reporter</td>
<td>Input parameter. An object providing the report method, which the component can use to write events to the event log.</td>
</tr>
</tbody>
</table>

### Sample Custom Java Components

For samples of the implementation of the custom components, see the following subdirectory of the installation directory:

`samples\SDK\ExternalParameters\Java_SDK\Java`
Developing a Custom Component in C or C++

1. Create a C or C++ project.
2. Add the following files to the project:
   - General.c
   - UTILS.c

   You can find the files in the following directory:
   
   `<INSTALL_DIR>/DataTransformation/samples/SDK/ExternalParameters/Cpp_SDK/Cpp`

3. Include all `*.h` files from the following directories:
   
   `<INSTALL_DIR>/DataTransformation/samples/SDK/ExternalParameters/Cpp_SDK/Cpp/include` 
   `<INSTALL_DIR>/DataTransformation/api/lib`

4. Set the linker option to add the following subdirectory:
   
   `<INSTALL_DIR>/DataTransformation/api/lib`

5. Create a module that implements the appropriate functions:

<table>
<thead>
<tr>
<th>Component Type</th>
<th>Input Type</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document processor</td>
<td>File</td>
<td>CMXProcessFile</td>
</tr>
<tr>
<td>Document processor</td>
<td>Multiple files</td>
<td>CMXProcessMultipleFiles</td>
</tr>
<tr>
<td>Document processor</td>
<td>Buffer</td>
<td>CMXProcessBuffer</td>
</tr>
<tr>
<td>Document processor</td>
<td>C++ stream</td>
<td>CMXProcessStream</td>
</tr>
<tr>
<td>Transformer</td>
<td>Null-terminated string</td>
<td>CMXTransformBuffer</td>
</tr>
<tr>
<td>Transformer</td>
<td>Buffer input that is not null-terminated</td>
<td>CMXTransformBinaryBuffer</td>
</tr>
</tbody>
</table>

There are some restrictions on whether a single module can implement more than one of the above interfaces. For more information about the interfaces, see the External-Component C and C++ Interface Reference.

6. For use on Windows platforms, compile the project to a DLL. For use on Linux or UNIX platforms, compile to a shared object.

7. Store the DLL or the shared object in the `<INSTALL_DIR>/DataTransformation/externLibs\user directory on every computer where you plan to use the component in a service.`
8. Create a script file that defines the display name of the component and its properties. Store the file in the `autoInclude\user directory.

   For more information about this step, see “Configuring a Custom Component” on page 316.

   You can then use the external component in transformations.

C/C++ Property Value Length Limitation

The property values that the script can pass to a C or C++ custom component can have lengths of up to 4000 characters.

C/C++ Interface Example

As an example of one of the C/C++ interfaces, consider a document processor that accepts file input. The processor must implement the `CMXProcessFile` function, which has the following syntax:

```c
int CMXProcessFile(
    void* sessionToken,
    const CMXContext* params,
    const IFfile_char_t* in_file,
    int in_len,
    IFfile_char_t** out_file,
    int* out_len,
    CMXEventReporter* reporter)
```

The following table describes the parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sessionToken</td>
<td>Input parameter. A pointer to the current session.</td>
</tr>
<tr>
<td>params</td>
<td>Input parameter. A structure containing the properties that the script passes to the component.</td>
</tr>
<tr>
<td>in_file</td>
<td>Input parameter. The full path of the file upon which the component should operate.</td>
</tr>
<tr>
<td>in_len</td>
<td>Input parameter. The length, in bytes, of the input file path.</td>
</tr>
<tr>
<td>out_file</td>
<td>Output parameter. The full path of a file that contains the output of the component.</td>
</tr>
<tr>
<td>out_len</td>
<td>Output parameter. The length, in bytes, of the output file path.</td>
</tr>
<tr>
<td>reporter</td>
<td>Input parameter. Provides the <code>report</code> method, which the component can use to write events to the event log.</td>
</tr>
<tr>
<td>Return value</td>
<td>Output parameter. 1 if successful, 0 if unsuccessful.</td>
</tr>
</tbody>
</table>

Sample Custom C/C++ Components

For online samples of the implementation, see the following subdirectory of the installation directory:

```
samples\SDK\ExternalParameters\Cpp_SDK\Cpp
```
The directory contains the following samples:

<table>
<thead>
<tr>
<th>Sample</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor.c</td>
<td>A document processor accepting either file or buffer input.</td>
</tr>
<tr>
<td>Transformer.c</td>
<td>A transformer accepting null-terminated string input.</td>
</tr>
</tbody>
</table>

## Configuring a Custom Component

After you develop a custom component, you must prepare a script file that defines the component. You cannot prepare the TGP file in the IntelliScript editor. Instead, you must prepare it in a text editor.

After you install the component and the TGP file, you can configure the custom component in the IntelliScript editor.

1. Create a text file and save it with a *.tgp extension.
   
   **Note:** You can define more than one external component in a single TGP file.

2. For each property that your external component supports, add lines such as the following to the TGP file:

   ```
   profile <CustomPropertyName> ofPT <DataType>
   {
       paramName = "<CustomPropertyName>" ;
   }
   
   <CustomPropertyName> is the name of a property that you want to display in the IntelliScript editor. <DataType> is the data type of the property. The supported data types are NamedParamIntT for an integer property, NamedParamBoolT for a boolean property, NamedParamStringT for a string property, or NamedParamListT for a property that is a list of strings.
   
3. For each external component that you wish to define, add lines such as the following to the TGP file. For a Java component:

   ```
   profile <ExternalComponentName> ofPT <ComponentType>
   {
       jclass = "<ClassName>" ;
       param1 = <CustomPropertyName1>();
       param2 = <CustomPropertyName2>();
   }
   ```

   For a C or C++ component:

   ```
   profile <ExternalComponentName> ofPT <ComponentType>
   {
       import_dll = DllPath("<DllName>") ;
       param1 = <CustomPropertyName1>();
       param2 = <CustomPropertyName2>();
   }
   ```
<ExternalComponentName> is the name of the external component that you want to display in the IntelliScript editor. <ComponentType> is one of the following values:

<table>
<thead>
<tr>
<th>For</th>
<th>ComponentType</th>
</tr>
</thead>
</table>
| A Java document processor with 0 to 4 properties | ExternalJavaProcessorNoParamsT  
ExternalJavaProcessor1ParamsT  
ExternalJavaProcessor2ParamsT  
... |
| A C or C++ document processor with 0 to 5 properties | ExternalProcessorNoParamsT  
ExternalProcessor1ParamsT  
ExternalProcessor2ParamsT  
... |
| A Java transformer with 0 to 10 properties | ExternalJavaTransformerNoParamsT  
ExternalJavaTransformer1ParamsT  
ExternalJavaTransformer2ParamsT  
... |
| A C or C++ transformer with 0 to 10 properties | ExternalTransformerNoParamsT  
ExternalTransformer1ParamsT  
ExternalTransformer2ParamsT  
... |

<ClassName> is the fully qualified name of the Java class. On Windows, <DllName> is the name of the DLL, without the *.dll extension. On Linux or UNIX, it is the name of the shared object, without the lib prefix or the *.so, extension.

<CustomPropertyName1> and <CustomPropertyName2> are the names of the properties that you configured in step 2.

5. Store the file in the DataTransformation\autoInclude\user subdirectory of the installation directory of every computer where you want to use the component.
6. If the Developer tool is open, close it and re-open it.
7. If an autoInclude error is displayed, review the TGP file for syntax errors or naming inconsistencies, and open the Developer tool again.
8. Open a project and insert the custom component in the script. The custom component name, which you assigned in step 3 above, appears in the IntelliScript drop-down list. The IntelliScript editor displays its properties.

**Sample Scripts Containing Custom Components**

You can find samples of script files that contain custom components in the following subdirectories of the installation directory:

samples\SDK\ExternalParameters\Java_SDK\autoInclude  
samples\SDK\ExternalParameters\Cpp_SDK\autoInclude
INDEX

A

AbsURL
  component 159
AcroForms
  processing PDF forms 71
actions
  compared to transformers 193
defining 193
  input and output 192
  properties of 193, 292
  side effects 192
AddEmptyTagsTransformer
  component 159
AddEventAction
  component 194
AddHeaderModifier
  component 284
AdditionalInputPort
  component 58
AdditionalOutputPort
  component 60
AddString
  component 160
AddStringModifier
  component 285
advanced properties
description 48
AFPToXML
  component 67
AggregateValues
  component 195
AllStructure
  component 148
AllStructureLocal
  component 149
alternative parsers
  selecting 119
AlternativeMappings
  component 248
Alternatives
  component 118
AlternativeSerializers
  component 233
AlternativeValidators
  component 293
anchor, repeating group
  highlighting all iterations 51
  anchors
  defining 108
  extent of complex 117
  location in IntelliScript 108
  Marker and Content 106
  phase 111
  properties of 110
  reference 117
relation to delimiters 107
relation to XML 107
serialization 226, 229
using transformers 156
AppendListItems
  component 197
AppendValues
  component 198
arithmetic
  computations 199
assigning
  value to output 219
attributes
data holders 92
AttributeSearch
  component 144
autoInclude
custom components 316

B

Base64Decoder
  component 160
Base64Encoder
  component 161
BidiConvert
  component 161
BinaryFormat
  component 81
BIRT
  XmlToDocument report generator 73
  XmlToDocument_372 report generator 73
buffer input port
  Data Processor transformation 6
buffer output port
  Data Processor transformation 7

C

C/C++
custom components 314
CalculateValue
  component 199
CDATADecode
  component 162
CDATAEncode
  component 162
ChangeCase
  component 163
ChoiceStructure
  component 150
ChoiceStructureLocal
  component 150
D

data validating 292
data holders
  destroying occurrences 104
  identifying source and target 258
  indexing multiple-occurrence 253
  mixed content 96
  single or multiple occurrence 103
Data Processing transformation
  startup component 10
Data Processor transformation
denormalized relational output 9
Data Processor Events view
  Data Processor transformation 18
  viewing event log 20
Data Processor functions
description 34
Data Processor Hex Source view
description 5
Data Processor transformation
  encoding settings 11
  creating 20
    description 4, 5
  exporting as a service 23
  input ports 6
  mapping XML to output ports 9
  normalized relational output 8
  output control settings 14
  output ports 7
  pivoted output 9
  ports 5
  processing settings 15
  relational output
    Data Processor transformation 7
    service parameter ports 6
    settings 11
  testing in the Data Viewer 23
  user logs 20
  views 5
  XML settings 16
Data Transformation service
  importing to the Model repository 24
  running from the command line 42
  data types
    searching for 116
  DateAddICU
    component 204
  DateDiff
    component 205
  DateDiffICU
    component 205
  DateFormat
    component 165
  DateFormatICU
    component 164

choose hierarchy
  Data Processor transformation 9
code pages
  transforming 168
  XSD schema 94
colors
  highlighting in example source document 50
combinations
  of lists 201
CombineValues
  component 201
CommaDelimited
  component 87
command-line interface
  CM_console command 42
complex segments
  streamer 270
ComplexSegment
  component 277
ComplexXmlSegment
  component 277
component properties
  values, description 49
Component view
  Data Processor transformation 5
component, global
  defining 47
component, local
  defining 47
component, script
  custom Java, developing 312
  description 46
component, startup in script
  description 49
components, custom
  C/C++ 314
components, custom script
  description 311
  properties 312
components, script
  description 46
  names 47
  properties, description 48
concatenation
  strings 197
condition
  ensuring in source document 208
Connect
  component 151
Content
  component 120
content anchors
  highlighting in example source document 50
ContentSerializer
  component 229, 234
CreateGuid
  component 164
CreateList
  component 202
CreateUUID
  component 164
custom component
  example 311
  Java, developing 312
custom components
  C/C++ 314
custom script components
  description 311
dates
  format of 164
Default statement
  XMap editor 27
default transformers
  in format 157
DelimitedSections
  component 123
DelimitedSectionsSerializer
  component 235
Delimiter
  component 87
DelimiterHierarchy
  component 88
delimiters
  custom hierarchy 82
  relation to anchors 107
denormalized relational output
  Data Processor transformation 9
derived data types
  XSI type 97
design-time event log
  description and location 19
direction property
  of anchors 110
DocList
  component 63
document processors
  custom C++ 70
  custom Java 70
  defining 66
  reference 67
  running multiple 73
document, example source
  description 50
DoNothingModifier
  component 286
Dos96HebToAscii
  component 167
DownloadFileToDataHolder
  component 206
dp:as_XML
  Data Processor function 34
dp:get_id
  Data Processor function 34
dp:lookup
  Data Processor function 34
dp:output
  Data Processor function 34
DumpValues
  component 207
dynamic
  offset 146
  search 148
DynamicTable
  component 167

EmbeddedMapper
  component 249
EmbeddedParser
  component 125
EmbeddedSerializer
  component 237
EmbeddedStructure
  component 152
  enclosed
  group 127
EnclosedGroup
  component 127
EnclosingDelimiters
  component 88
EncodeAsUrl
  component 168
Encoder
  component 168
  encoding
  code page transformer 168
  XSD schema 94
  encoding guidelines
  Data Processor transformation 14
  encoding settings
  Data Processor transformation 11
EnsureCondition
  component 208
entering characters
  non-keyboard 49
  entry point, script
  description 49
Enumeration
  component 295
errors
  failure handling 289
event log
  custom events 194
  viewing 20
  event log, design-time
  description and location 19
event types
  Data Processor transformation 17
events
  Data Processor transformation 17
  handling 288
Events view
  Data Processor transformation 18
Events view, Data Processor
  viewing event log 20
example input file
  Data Processor transformation 6
example source
  description 50
  setting 50
  viewing 51
XMap Schema panel 26
Example Source
  creating in Data Processor transformation 22
example source document, highlighting
  description 50
example_source property
  mapper 246
  serializer 231
Excel
  generating from XML 74
  parsing as XML 68
ExcelToDataXml
  component 68
ExcelToXml component 68
ExcludedItems component 210
ExpandFrameSet component 69
ExternalJavaPreProcessor component 70
ExternalPreProcessor component 70
ExternalTransformer component 169
extracting content
Content anchor 120
ExtractRecord component 129

F
failure
effect on parent 289
failure event
Data Processor transformation 17
failure handling
variables for 100
failures
handing 289
handling 288
fatal error event
Data Processor transformation 17
file input port
Data Processor transformation 6
file output port
Data Processor transformation 7
FileSearch component 63
FindReplaceAnchor component 129
footer segment streamer 270
format
preprocessors 91
FormatNumber component 170
forms
processing PDF 71
frameset
 parsing HTML 69
FromFloat component 171
FromInteger component 171
FromPackDecimal component 172
FromSignedDecimal component 172

G
global component
defining 47
grid
XMap 27
group
performing actions on 132
repeating 137

Group component 132
Group statements
XMap 29
GroupMapping component 250
GroupSerializer component 238

H
handling failures 289
header segment streamer 270
Hebrew
code-page conversion 173
hebrewBidi component 173
HebrewDosToWindows component 173
HebrewEBCDICOldCodeToWindows component 173
hebUniToAscii component 173
hebUtf8ToAscii component 173
Hex Source view, Data Processor
description 5
Hexadecimal Source view
description 51
hidden properties showing 48
highlighting in example source document
description 50
HIPAA validation 70
HIPAAValidator component 70
HL7 component 89
HTML
removing tags 181
transforming entities 174
HtmlEntitiesToASCII component 174
HtmlFormat component 83
HtmlProcessor component 91, 174

I
indexing
example 255
multiple-occurrence data holders 104
initialization
variables 102
InjectFP component 174
InjectString component 175
InlineTable component 175
Input Expression Editor
Data Processor transformation 33
InputPort
  component 63
Intelli mode
description 51
IntelliScript
defining anchors in 109
IntelliScript editor
description 51
IntelliScript Help view
description 51
invalid data
detecting 288
iterations
  RepeatingGroup anchor 137

J
Java
custom component, developing 312
JavaScript
  extensions 209
  syntax reference 208
JavaTransformer
  component 176

K
key
  properties of 264
Key
  component 264

L
LearnByExample
  component 145
LengthEquals
  component 296
list data types
  attributes 104
list types
  mapping to 116
lists
  combining 201
  creating 202
  multiple-occurrence data holders 103
  of variables 103
  sorting 220
local component
defining 47
LocalFile
  component 64
LocalFile example source
description 50
locations
  marking in source document 134
Locator
  component 266
LocatorByKey
  component 267
LocatorByOccurrence
  component 268
locators
  properties of 264
log
definition 18
log, design-time event
description and location 19
logs
  writing to 288, 307
LookupTransformer
  component 176
loop
  RepeatingGroup anchor 137

M
Map
  component 211
mapper
calling secondary 249
  creating 243
  input validation 98
Mapper
  component 246
  description 4
mapper anchors
  properties of 245
  reference 247
mappers
  properties of 245
  running in parser 214, 218
  using indexing 255
mapping statements
  XMap editor 27
Marker
  component 134
marker anchors
  highlighting in example source document 50
markers
  in streamers 272
MarkerStreamer
  component 278
marking property
  of anchors 110, 278
MaxLength
  component 297
MaxNumber
  component 298
messages
  warning notifications 307
MinLength
  component 298
MinNumber
  component 299
missing data
  failure handling 289
missing text
  searching by optional Group 133
mixed content
  data holders in 96
  in schema 94
  mapping to 108
multiple occurrence
  data holders 103
  destroying occurrences 104
  variables 103
multiple-occurrence data holders
  combining 201
  creating lists in 202
  indexing 253
mapping anchors to 108
sorting 220

N
names 47
NewlineSearch 146
non-keyboard characters 49
NormalizeClosingTags 178
normalized relational output 8
Data Processor transformation 178
Notification component 308
notification event 17
Data Processor transformation 17
notification events 143
StructureDefinition 308
NotificationGroupName 308
NotificationHandler component 309
notifications 146
generating 288
triggering 212
writing messages 307
Notify 212
NotifyFailure component 310
NumberEquals 300
numbers 170

O
offset 146
dynamically defined 146
OffsetSearch 146
on fail 28
Option statement 27
Option statements 30
optional failure 289
effect on parent 289
optional failure event 17
Data Processor transformation 17
optional property 289
failure handling 289
setting 290
output control settings 14
Data Processor transformation 14
OutputDataHolder 224
OutputFile 224
OutputPort 64

P
packed decimals 172
parameters 102
passing to transformation 102
parser 54
Parser 55
description 4
parsers 125, 237
calling secondary 215
running secondary 215
path 159
resolving relative 159
pattern matching 180
patterns 270
segment opening and closing 270
PatternSearch 146
PDF 71
processing PDF forms 71
PDF conversion 76
PDF files 71
using PdToTxt_4 processor 75
PDF output 73
XmlToDocument postprocessor 73
XmlToDocument postprocessor_372 73
PDF support 71, 72
converting PDF files 71, 72
PdfFormToXml_1_00 71
PdfToTxt_3_02 71
PdfToTxt_4 72
using 75
performance issues 100
VarLastFailure variable 289
phase 111
of anchor search 111
phase property 110
of anchors 110
phases 112
nested 112
pivot relational output 9
Data Processor transformation 9
platform independence 54
parsers 54
Ports view 9
Data Processor transformation 9
Positional 89
component 89
postprocessor 73
XmlToDocument 73
XmlToDocument_372 73
PostScript 90
component 90
PowerpointToTextML 72
component 72
pre-processors 66
defining 66
predicate statement 33
XPath expressions 33
preprocessors
  document 66
  format 91
ProcessByTransformers
  component 73
ProcessorPipeline
  component 73
processors
  custom C++ 70
  custom Java 70
  document 66
  reference 67
  using transformers as 157
properties
  custom script components 312
  of actions 193, 292
  of anchors 110
  of mappers 245
  of serializers 231
  of streamers 276
  script components, description 48
  transformers 158
properties, advanced
  description 48
properties, component
  values, description 49
properties, hidden
  showing 48
properties, simple
  description 48

R
  records
    extracting and validating 140
    extracting in StructureDefinition 129
RecordStructure
  component 152
RecordStructureLocal
  component 153
reference
  anchors 117
  delimiters 86
  document processors 67
  format preprocessors 91
  formats 80
  indexing 264
  mapper anchors 247
  mappers 246
  parsers 55
  serialization anchors 232
reference points
  around anchors 110, 278
  Marker anchor 134
  of search scope 114
References view
  Data Processor transformation 11
regex
  regular expressions 180
  regular expressions
    syntax 180
RegularExpression
  component 179
RemoveMarginSpace
  component 181
RemoveRtfFormatting
  component 181
RemoveTags
  component 181
repeating group
  highlighting in example source document 50
repeating group anchor
  highlighting all iterations 51
Repeating Group statements
  description 30
  XMap editor 27
repeating segment
  streamer 270
RepeatingGroup
  component 137
RepeatingGroupMapping
  component 251
RepeatingGroupSerializer
  component 239
Replace
  component 182
  replacing text
    in source document 129
report generator
  BIRT 73
Repository view
  Data Processor transformation 5
ResetVisitedPages
  component 213
Resize
  component 183
ResultFile
  component 225
retrieving content
  Content anchor 120
ReverseTransformer
  component 183
right-to-left text
  reversing 161, 173
rollback
  after failure 289
Router statements
  XMap editor 27, 30
RTF
  component 90
RtfFormat
  component 84
RtfProcessor
  component 91, 184
RtfToASCII
  component 184
RtfToTextML
  component 73
Run XMap statement
  mapping statement 31
run-time event log
  Data Processor transformation 19
RunMapper
  component 214
RunParser
  component 215
RunSerializer
  component 217
RunXMapper
  component 218
sample script
  importing 53
sample source file
  defining in Data Processor transformation 21
schema
  encoding 94
schemas
  Data Processor transformation 11
  editors 94
  included schemas 94
  IntelliJScript representation 96
sample XML 93
unsupported features 95
validation 97
XSD files 92
script
  description 45
  editor, IntelliJScript 51
importing sample 53
samples 52
structure 46
script component
  custom Java, developing 312
  description 46
script components
  description 46
  names 47
properties, description 48
script components, custom
  description 311
  properties 312
Script Help view
  Data Processor transformation 5
script mode
  description 51
script object
  creating in Data Processor transformation 21
search
  anchor direction 110
  dynamically defined search string 148
search criteria
  for anchors 112
search scope
  adjusting 114
  for anchors 112
searcher
  components 115, 144
secondary mapper
  EmbeddedMapper anchor 249
secondary parser
  EmbeddedParser anchor 125, 237
segments
  processing in streamer 270
SegmentSearch
  component 147
select-and-click
  defining anchors 109
SequenceStructure
  component 154
SequenceStructureLocal
  component 154
serialization
  mode 227
  using transformers in 158
serialization anchors
  properties of 231
reference 232
sequence of operation 230
serializer
  controlling auto-generation 227
  input validation 98
Serializer
  component 231
serializers
  creating from parser 226
  properties of 231
  running in parser 217
troubleshooting auto-generated 228
service name
  variable storing 100
service parameter port
  Data Processor transformation 6
service parameter ports
  Data Processor transformation 6
service parameters
  passing to transformation 102
service, Data Transformation
  running from the command line 42
ServiceLocation
  variable 99
Settings view
  Data Processor transformation 11
SetValue
  component 219
SGML
  component 90
signed decimals
  numbers 172
simple properties
  description 48
SimpleSegment
  component 280
SimpleXmlSegment
  component 280
single occurrence
  data holders 103
skip missing input
  XMap statement 28
Sort
  component 220
sorting
  multiple-occurrence data holders 220
source
  property 258, 259
source document, highlighting
  description 50
SpaceDelimited
  component 90
splitting
  files 222
  splitting large inputs
  streamer 269
startup component
  Data Processor transformation 10
startup component, script
  description 49
streamer
  complex segments 270
  creating 272
  footer segment 270
  header concatenation 271
  header segment 270
  output 271
  repeating segment 270
segment opening and closing patterns 270
splitting large inputs 269
Streamer
component 281
description 4
streamers
properties of 276
StreamerVariable
component 282
strings
concatenating 197, 198
StringSerializer
component 229
structured parsing
StructureDefinition
component 140
extracting records 129
SubString
component 184
synchronize with editor
Data Processor transformation 23
system
variables 99
system time
variable 100

T
TabDelimited
component 90
table configuration editor
PdfToTxt_4 75
tables
processing PDF 75
target
property 258, 262
test expression
Input XPath Expression Editor 33
Text
component 64
Text example source
description 50
text streamer
principle of operation 270
TextFormat
component 84
TextML
XML schema 74
TextSearch
component 147
TGP file
for custom components 316
time
system 100
times
format of 164
ToFloat
component 185
ToInteger
component 185
ToPackDecimal
component 186
TransformationStartTime
component 187
TransformByParser
component 187
TransformByProcessor
component 188
TransformByService
component 189
Transformer
description 4
TransformerPipeline
component 190
transformers
as document preprocessors 157
compared to actions 193
custom DLL 169
custom Java 176
default 157
defining 156
in serialization 158
properties of 158
sequences of 157
using as document processors 73
using in anchors 156
types
XSI 97
TypeSearch
component 148

U
UNIX
designing parsers for 54
URL
relative to absolute 159
URL example source
description 50
user log
Data Processor transformation 20
variable defining location 100

V
ValidateByExpression
component 301
ValidateByPattern
component 302
ValidateByTransformer
component 303
ValidateByType
component 304
ValidateDate
component 305
ValidateValue
component 220
validation
XML 97
XML input 98
XML parser output 98
Validation Rule Language
VRL 220
ValidatorPipeline
component 306
validators
input data 292
values, component properties
description 49
VarCurrentPost
variable 100
VarCurrentURL
variable 100
<table>
<thead>
<tr>
<th>Variable component</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable component</td>
<td>102</td>
</tr>
<tr>
<td>variables</td>
<td></td>
</tr>
<tr>
<td>data holders</td>
<td>92</td>
</tr>
<tr>
<td>in streamers</td>
<td>272</td>
</tr>
<tr>
<td>initialization</td>
<td>102</td>
</tr>
<tr>
<td>lists</td>
<td>103</td>
</tr>
<tr>
<td>mapping anchors to</td>
<td>102, 108</td>
</tr>
<tr>
<td>system</td>
<td>99</td>
</tr>
<tr>
<td>user-defined</td>
<td>99</td>
</tr>
<tr>
<td>using in actions</td>
<td>102</td>
</tr>
<tr>
<td>XMap editor</td>
<td>36</td>
</tr>
<tr>
<td>VarLastFailure</td>
<td></td>
</tr>
<tr>
<td>variable</td>
<td>100</td>
</tr>
<tr>
<td>VarLinkURL</td>
<td></td>
</tr>
<tr>
<td>variable</td>
<td>99</td>
</tr>
<tr>
<td>VarRequestedURL</td>
<td></td>
</tr>
<tr>
<td>variable</td>
<td>100</td>
</tr>
<tr>
<td>VarServiceInfo</td>
<td></td>
</tr>
<tr>
<td>variable</td>
<td>100</td>
</tr>
<tr>
<td>VarSystem</td>
<td></td>
</tr>
<tr>
<td>system time</td>
<td>100</td>
</tr>
<tr>
<td>view</td>
<td></td>
</tr>
<tr>
<td>Hexadecimal Source</td>
<td></td>
</tr>
<tr>
<td>description</td>
<td>51</td>
</tr>
<tr>
<td>view</td>
<td></td>
</tr>
<tr>
<td>IntelliScript Help</td>
<td></td>
</tr>
<tr>
<td>description</td>
<td>51</td>
</tr>
<tr>
<td>VRL</td>
<td></td>
</tr>
<tr>
<td>Validation Rule Language</td>
<td>220</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>warning event</td>
<td></td>
</tr>
<tr>
<td>Data Processor transformation</td>
<td>17</td>
</tr>
<tr>
<td>WellFormedModifier</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>286</td>
</tr>
<tr>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>parsing as XML</td>
<td>73</td>
</tr>
<tr>
<td>WordToXml</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>73</td>
</tr>
<tr>
<td>WriteSegment</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>287</td>
</tr>
<tr>
<td>WriteValue</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>221</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xerces</td>
<td></td>
</tr>
<tr>
<td>XML validation</td>
<td>98</td>
</tr>
<tr>
<td>XMap</td>
<td></td>
</tr>
<tr>
<td>description</td>
<td>25</td>
</tr>
<tr>
<td>Group statements</td>
<td>29</td>
</tr>
<tr>
<td>Router statements</td>
<td>30</td>
</tr>
<tr>
<td>Schema panel</td>
<td>26</td>
</tr>
<tr>
<td>statement types</td>
<td>27</td>
</tr>
<tr>
<td>variables</td>
<td>36</td>
</tr>
<tr>
<td>XMap editor</td>
<td></td>
</tr>
<tr>
<td>grid</td>
<td>27</td>
</tr>
<tr>
<td>XMap object</td>
<td></td>
</tr>
<tr>
<td>creating in Data Processor transformation</td>
<td>22</td>
</tr>
<tr>
<td>XMap statement</td>
<td></td>
</tr>
<tr>
<td>field descriptions</td>
<td>28</td>
</tr>
<tr>
<td>XML</td>
<td></td>
</tr>
<tr>
<td>adding empty tags</td>
<td>159</td>
</tr>
<tr>
<td>mapping anchors to</td>
<td>107</td>
</tr>
<tr>
<td>schemas</td>
<td>92</td>
</tr>
<tr>
<td>validation</td>
<td>98</td>
</tr>
<tr>
<td>XSLT transformation</td>
<td>190</td>
</tr>
<tr>
<td>XML attributes</td>
<td></td>
</tr>
<tr>
<td>data holders</td>
<td>92</td>
</tr>
<tr>
<td>XML elements</td>
<td></td>
</tr>
<tr>
<td>data holders</td>
<td>92</td>
</tr>
<tr>
<td>XML schemas</td>
<td></td>
</tr>
<tr>
<td>sample</td>
<td>93</td>
</tr>
<tr>
<td>XML Spy</td>
<td></td>
</tr>
<tr>
<td>schema editor</td>
<td>94</td>
</tr>
<tr>
<td>XML streamer</td>
<td></td>
</tr>
<tr>
<td>principle of operation</td>
<td>274</td>
</tr>
<tr>
<td>XmapFormat</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>85</td>
</tr>
<tr>
<td>XMLLookupTable</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>190</td>
</tr>
<tr>
<td>XmlSegment</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>283</td>
</tr>
<tr>
<td>XmlStreamer</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>283</td>
</tr>
<tr>
<td>XmlToDocument</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>73</td>
</tr>
<tr>
<td>XmlToDocument_372</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>73</td>
</tr>
<tr>
<td>XmlToExcel</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>74</td>
</tr>
<tr>
<td>XPath</td>
<td></td>
</tr>
<tr>
<td>modified notation</td>
<td>96</td>
</tr>
<tr>
<td>XPath Expression Editor</td>
<td></td>
</tr>
<tr>
<td>Data Processor transformation</td>
<td>33</td>
</tr>
<tr>
<td>XPath expressions</td>
<td></td>
</tr>
<tr>
<td>XMap</td>
<td>32</td>
</tr>
<tr>
<td>XSD</td>
<td></td>
</tr>
<tr>
<td>editors</td>
<td>92</td>
</tr>
<tr>
<td>schema editors</td>
<td>94</td>
</tr>
<tr>
<td>schema encoding</td>
<td>94</td>
</tr>
<tr>
<td>unsupported schema features</td>
<td>95</td>
</tr>
<tr>
<td>XML schema files</td>
<td>93</td>
</tr>
<tr>
<td>XSD files</td>
<td></td>
</tr>
<tr>
<td>schemas</td>
<td>92</td>
</tr>
<tr>
<td>XSI types</td>
<td></td>
</tr>
<tr>
<td>mapping data holders</td>
<td>97</td>
</tr>
<tr>
<td>XSLT</td>
<td></td>
</tr>
<tr>
<td>running transformations</td>
<td>223</td>
</tr>
<tr>
<td>XSLTMap</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>223</td>
</tr>
<tr>
<td>XSLTTransformer</td>
<td></td>
</tr>
<tr>
<td>component</td>
<td>190</td>
</tr>
</tbody>
</table>