Informatica

Informatica® B2B Data Transformation
10.2 HF1

Data Transformation User Guide

Informatica B2B Data Transformation Data Transformation User Guide
10.2 HF1
July 2018

© Copyright Informatica LLC 2001, 2018

This software and documentation contain proprietary information of Informatica LLC and are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright law. Reverse engineering of the software is prohibited. No part of this document may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording or otherwise) without prior consent of Informatica LLC. This Software may be protected by U.S. and/or international Patents and other Patents Pending.

Use, duplication, or disclosure of the Software by the U.S. Government is subject to the restrictions set forth in the applicable software license agreement and as provided in DFARS 227.7202-1(a) and 227.7702-3(a) (1995), DFARS 252.227-7013©(1)(ii) (OCT 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 52.227-14 (ALT III), as applicable.

The information in this product or documentation is subject to change without notice. If you find any problems in this product or documentation, please report them to us in writing.

Informatica, Informatica Platform, Informatica Data Services, PowerCenter, PowerCenterRT, PowerCenter Connect, PowerCenter Data Analyzer, PowerExchange, PowerMart, Metadata Manager, Informatica Data Quality, Informatica Data Explorer, Informatica B2B Data Transformation, Informatica B2B Data Exchange Informatica On Demand, Informatica Identity Resolution, Informatica Application Information Lifecycle Management, Informatica Complex Event Processing, Ultra Messaging, Informatica Master Data Management, and Live Data Map are trademarks or registered trademarks of Informatica LLC in the United States and in jurisdictions throughout the world. All other company and product names may be trade names or trademarks of their respective owners.

Portions of this software and/or documentation are subject to copyright held by third parties, including without limitation: Copyright DataDirect Technologies. All rights reserved. Copyright © Sun Microsystems. All rights reserved. Copyright © RSA Security Inc. All Rights Reserved. Copyright © Ordinal Technology Corp. All rights reserved. Copyright © Aandacht c.v. All rights reserved. Copyright Genivia, Inc. All rights reserved. Copyright Isomorphic Software. All rights reserved. Copyright © Meta Integration Technology, Inc. All rights reserved. Copyright © Intalio. All rights reserved. Copyright © Oracle. All rights reserved. Copyright © Adobe Systems Incorporated. All rights reserved. Copyright © DataArt, Inc. All rights reserved. Copyright © ComponentSource. All rights reserved. Copyright © Microsoft Corporation. All rights reserved. Copyright © Rogue Wave Software, Inc. All rights reserved. Copyright © Teradata Corporation. All rights reserved. Copyright © Yahoo! Inc. All rights reserved. Copyright © Glyph & Cog, LLC. All rights reserved. Copyright © Thinkmap, Inc. All rights reserved. Copyright © Clearpace Software Limited. All rights reserved. Copyright © Information Builders, Inc. All rights reserved. Copyright © OSS Nokalva, Inc. All rights reserved. Copyright Edifecs, Inc. All rights reserved. Copyright Cleo Communications, Inc. All rights reserved. Copyright © International Organization for Standardization 1986. All rights reserved. Copyright © ej-technologies GmbH. All rights reserved. Copyright © Jaspersoft Corporation. All rights reserved. Copyright © International Business Machines Corporation. All rights reserved. Copyright © yWorks GmbH. All rights reserved. Copyright © Lucent Technologies. All rights reserved. Copyright © University of Toronto. All rights reserved. Copyright © Daniel Veillard. All rights reserved. Copyright © Unicode, Inc. Copyright IBM Corp. All rights reserved. Copyright © MicroQuill Software Publishing, Inc. All rights reserved. Copyright © PassMark Software Pty Ltd. All rights reserved. Copyright © LogiXML, Inc. All rights reserved. Copyright © 2003-2010 Lorenzi Davide, All rights reserved. Copyright © Red Hat, Inc. All rights reserved. Copyright © The Board of Trustees of the Leland Stanford Junior University. All rights reserved. Copyright © EMC Corporation. All rights reserved. Copyright © Flexera Software. All rights reserved. Copyright © Jinfonet Software. All rights reserved. Copyright © Apple Inc. All rights reserved. Copyright © Telerik Inc. All rights reserved. Copyright © BEA Systems. All rights reserved. Copyright © PDFlib GmbH. All rights reserved. Copyright © Orientation in Objects GmbH. All rights reserved. Copyright © Tanuki Software, Ltd. All rights reserved. Copyright © Ricebridge. All rights reserved. Copyright © Sencha, Inc. All rights reserved. Copyright © Scalable Systems, Inc. All rights reserved. Copyright © jQWidgets. All rights reserved. Copyright © Tableau Software, Inc. All rights reserved. Copyright© MaxMind, Inc. All Rights Reserved. Copyright © TMate Software s.r.o. All rights reserved. Copyright © MapR Technologies Inc. All rights reserved. Copyright © Amazon Corporate LLC. All rights reserved. Copyright © Highsoft. All rights reserved. Copyright © Python Software Foundation. All rights reserved. Copyright © BeOpen.com. All rights reserved. Copyright © CNRI. All rights reserved.

This product includes software developed by the Apache Software Foundation (http://www.apache.org/), and/or other software which is licensed under various versions of the Apache License (the "License"). You may obtain a copy of these Licenses at http://www.apache.org/licenses/. Unless required by applicable law or agreed to in writing, software distributed under these Licenses is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licenses for the specific language governing permissions and limitations under the Licenses.

This product includes software which was developed by Mozilla (http://www.mozilla.org/), software copyright The JBoss Group, LLC, all rights reserved; software copyright © 1999-2006 by Bruno Lowagie and Paulo Soares and other software which is licensed under various versions of the GNU Lesser General Public License Agreement, which may be found at http:// www.gnu.org/licenses/lgpl.html. The materials are provided free of charge by Informatica, "as-is", without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

The product includes ACE(TM) and TAO(TM) software copyrighted by Douglas C. Schmidt and his research group at Washington University, University of California, Irvine, and Vanderbilt University, Copyright (©) 1993-2006, all rights reserved.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (copyright The OpenSSL Project. All Rights Reserved) and redistribution of this software is subject to terms available at http://www.openssl.org and http://www.openssl.org/source/license.html.

This product includes Curl software which is Copyright 1996-2013, Daniel Stenberg, <daniel@haxx.se>. All Rights Reserved. Permissions and limitations regarding this software are subject to terms available at http://curl.haxx.se/docs/copyright.html. Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

The product includes software copyright 2001-2005 (©) MetaStuff, Ltd. All Rights Reserved. Permissions and limitations regarding this software are subject to terms available at http://www.dom4j.org/ license.html.

The product includes software copyright © 2004-2007, The Dojo Foundation. All Rights Reserved. Permissions and limitations regarding this software are subject to terms available at http://dojotoolkit.org/license.

This product includes ICU software which is copyright International Business Machines Corporation and others. All rights reserved. Permissions and limitations regarding this software are subject to terms available at http://source.icu-project.org/repos/icu/icu/trunk/license.html.

This product includes software copyright © 1996-2006 Per Bothner. All rights reserved. Your right to use such materials is set forth in the license which may be found at http:// www.gnu.org/software/ kawa/Software-License.html.

This product includes OSSP UUID software which is Copyright © 2002 Ralf S. Engelschall, Copyright © 2002 The OSSP Project Copyright © 2002 Cable & Wireless Deutschland. Permissions and limitations regarding this software are subject to terms available at http://www.opensource.org/licenses/mit-license.php.

This product includes software developed by Boost (http://www.boost.org/) or under the Boost software license. Permissions and limitations regarding this software are subject to terms available at http:/ /www.boost.org/LICENSE_1_0.txt.

This product includes software copyright © 1997-2007 University of Cambridge. Permissions and limitations regarding this software are subject to terms available at http:// www.pcre.org/license.txt.

This product includes software copyright © 2007 The Eclipse Foundation. All Rights Reserved. Permissions and limitations regarding this software are subject to terms available at http:// www.eclipse.org/org/documents/epl-v10.php and at http://www.eclipse.org/org/documents/edl-v10.php.

This product includes software licensed under the terms at http://www.tcl.tk/software/tcltk/license.html, http://www.bosrup.com/web/overlib/?License, http://www.stlport.org/doc/ license.html, http://asm.ow2.org/license.html, http://www.cryptix.org/LICENSE.TXT, http://hsqldb.org/web/hsqlLicense.html, http://httpunit.sourceforge.net/doc/ license.html, http://jung.sourceforge.net/license.txt , http://www.gzip.org/zlib/zlib_license.html, http://www.openldap.org/software/release/license.html, http://www.libssh2.org, http://slf4j.org/license.html, http://www.sente.ch/software/OpenSourceLicense.html, http://fusesource.com/downloads/license-agreements/fuse-message-broker-v-5-3- license-agreement; http://antlr.org/license.html; http://aopalliance.sourceforge.net/; http://www.bouncycastle.org/licence.html; http://www.jgraph.com/jgraphdownload.html; http://www.jcraft.com/jsch/LICENSE.txt; http://jotm.objectweb.org/bsd_license.html; . http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231; http://www.slf4j.org/license.html; http://nanoxml.sourceforge.net/orig/copyright.html; http://www.json.org/license.html; http://forge.ow2.org/projects/javaservice/, http://www.postgresql.org/about/licence.html, http://www.sqlite.org/copyright.html, http://www.tcl.tk/software/tcltk/license.html, http://www.jaxen.org/faq.html, http://www.jdom.org/docs/faq.html, http://www.slf4j.org/license.html; http://www.iodbc.org/dataspace/iodbc/wiki/iODBC/License; http://www.keplerproject.org/md5/license.html; http://www.toedter.com/en/jcalendar/license.html; http://www.edankert.com/bounce/index.html; http://www.net-snmp.org/about/license.html; http://www.openmdx.org/#FAQ; http://www.php.net/license/3_01.txt; http://srp.stanford.edu/license.txt; http://www.schneier.com/blowfish.html; http://www.jmock.org/license.html; http://xsom.java.net; http://benalman.com/about/license/; https://github.com/CreateJS/EaselJS/blob/master/src/easeljs/display/Bitmap.js; http://www.h2database.com/html/license.html#summary; http://jsoncpp.sourceforge.net/LICENSE; http://jdbc.postgresql.org/license.html; http://protobuf.googlecode.com/svn/trunk/src/google/protobuf/descriptor.proto; https://github.com/rantav/hector/blob/master/LICENSE; http://web.mit.edu/Kerberos/krb5-current/doc/mitK5license.html; http://jibx.sourceforge.net/jibx-license.html; https://github.com/lyokato/libgeohash/blob/master/LICENSE; https://github.com/hjiang/jsonxx/blob/master/LICENSE; https://code.google.com/p/lz4/; https://github.com/jedisct1/libsodium/blob/master/LICENSE; http://one-jar.sourceforge.net/index.php?page=documents&file=license; https://github.com/EsotericSoftware/kryo/blob/master/license.txt; http://www.scala-lang.org/license.html; https://github.com/tinkerpop/blueprints/blob/master/LICENSE.txt; http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html; https://aws.amazon.com/asl/; https://github.com/twbs/bootstrap/blob/master/LICENSE; https://sourceforge.net/p/xmlunit/code/HEAD/tree/trunk/LICENSE.txt; https://github.com/documentcloud/underscore-contrib/blob/master/LICENSE, and https://github.com/apache/hbase/blob/master/LICENSE.txt.

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 new BSD License (http://opensource.org/licenses/BSD-3-Clause), the MIT License (http://www.opensource.org/licenses/mit-license.php), the Artistic License (http://www.opensource.org/licenses/artistic-license-1.0) and the Initial Developer's Public License Version 1.0 (http://www.firebirdsql.org/en/initial-developer-s-public-license-version-1-0/).

This product includes software copyright © 2003-2006 Joe Walnes, 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 product includes software Copyright (c) 2013 Frank Balluffi and Markus Moeller. All rights reserved. Permissions and limitations regarding this software are subject to terms of the MIT license.

See patents at https://www.informatica.com/legal/patents.html.

DISCLAIMER: Informatica LLC 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 LLC 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.

Publication Date: 2018-09-25

Table of Contents

Preface 20

Informatica Resources. 20

Informatica Network. 20

Informatica Knowledge Base. 20

Informatica Documentation. 20

Informatica Product Availability Matrixes. 21

Informatica Velocity. 21

Informatica Marketplace. 21

Informatica Global Customer Support. 21

Chapter 1: Introduction to Data Transformation. 22

Data Transformation Overview. 22

Data Transformation Process Architecture. 23

Data Transformation Components. 24

Chapter 2: Data Processor Transformation. 25

Data Processor Transformation Overview. 25

Data Processor Transformation Views. 26

Data Processor Transformation Ports. 27

Data Processor Transformation Input Ports. 27

Data Processor Transformation Output Ports. 28

Pass-Through Ports. 29

Startup Component. 29

References. 29

Data Processor Transformation Settings. 30

Character Encoding. 30

Rules and Guidelines for Character Encoding. 33

Output Settings. 33

Processing Settings. 34

XMap Settings. 35

XML Output Configuration. 35

Events. 37

Event Types. 37

Data Processor Events View. 37

Logs. 38

Design-Time Event Log. 38

Run-Time Event Log. 39

Viewing an Event Log in the Data Processor Events View. 39

User Log. 39

Data Processor Transformation Development. 40

Create the Data Processor Transformation. 40
Select the Schema Objects 41
Create Objects in a Blank Data Processor Transformation. 41
Create the Ports. 43
Testing the Transformation. 44
Data Processor Transformation Import and Export. 44
Exporting the Data Processor Transformation as a Service. 44
Importing Multiple Data Transformation Services. 45
Importing a Data Transformation Service 45
Exporting a Mapping with a Data Processor Transformation to PowerCenter. 46

Chapter 3: Wizard Input and Output Formats. 47

Wizard Input and Output Formats Overview. 47
ASN.1 Library. 47
ASN.1 Protocol Specification. 48
ASN.1 Input Message Structure. 49
Installing the ASN.1 Library. 49
Creating an ASN.1 Library Transformation. 50
Installing the ASN-1Step Utility. 52
ASN.1 Streamer. 52
AsnToXml Document Processor. 53
ASN.1 Transformers. 54
BCDSwapToString. 55
BCDToString. 55
HexSequenceToDecimal. 55
HexToDecimal. 56
Swap. 56
Avro. 56
Avro Input and Complex File Reader. 57
Avro Data Compression with the Snappy Codec. 58
Configure a Transformation with Avro Input. 58
Configure a Transformation with Avro Output. 60
COBOL Processing Library. 61
Creating a Transformation for COBOL. 61
COBOL Data Definitions. 62
Test Procedures. 62
Editing a Transformation for COBOL. 63
Optimizing Large COBOL File Processing in the Hadoop Environment. 63
JSON. 64
JSON Schemas. 64
Sample JSON Schema. 65
Creating a Transformation with JSON. 66
Parquet. 67

Creating a Transformation with Parquet Input or Output. . . . 67
Configure the Complex File Reader For Parquet Input. . . . 68
Configure a Transformation with Parquet Output. . . . 68
XML. . . . 69
Creating a Transformation that Transforms XML. . . . 69

Chapter 4: Relational Input and Output. . . . 70

Relational Input and Output Overview. . . . 70
Relational Input. . . . 70
Relational Input Port Configuration. . . . 71
Guidelines to Link Input Ports. . . . 72
Define Input Relational Ports with the Overview View. . . . 72
Clustering_Key Ports. . . . 73
Normalized Relational Input 74
Pivoted Relational Input. . . . 74
Denormalized Relational Input. . . . 75
Mapping Relational Ports to Hierarchical Nodes. . . . 75
Relational Output. . . . 76
Relational Output Port Configuration. . . . 76
Define Output Relational Ports with the Overview View. . . . 77
Normalized Relational Output. . . . 78
Pivoted Relational Output. . . . 78
Denormalized Relational Output. . . . 79

Chapter 5: Using the IntelliScript Editor. . . . 80

IntelliScript Editor Overview. . . . 80
Creating a Script. . . . 80
Opening an IntelliScript Editor. . . . 81
IntelliScript and Data Viewer. . . . 81
Finding Anchors. . . . 81
Components and Properties. . . . 81
Basic and Advanced Properties. . . . 82
Editing Procedures. . . . 82
Basic Procedure for Editing. . . . 82
Copy and Paste. . . . 82
Drag and Drop. . . . 83
Find and Replace. . . . 83
Inserting Components in the IntelliScript. . . . 83
Editing the Properties of a Component. . . . 83
Inserting Tabs, Newlines, and Other Special Characters. . . . 83
Defining a Global Component. . . . 84
Viewing Help About a Component. . . . 84
IntelliScript Icons. . . . 85

Saving the IntelliScript. . . . 85
IntelliScript Editor Menus. . . . 86

Chapter 6: XMap. . . . 88

XMap Overview. . . . 88
XMap Schemas. . . . 89
Mapping Statements. . . . 90
Mapping Statement Types. . . . 90
Map Statements. . . . 91
Group Statements. . . . 93
Repeating Group Statements. . . . 94
Router Statements. . . . 96
Option Statements. . . . 98
Default Statements. . . . 99
Run XMap Statements. . . . 100
RunMapplet Statement. . . . 101
MappletInput Statement. . . . 102
MappletOutput Statement. . . . 103
Creating an XMap. . . . 104
Using the XMap Editor Grid. . . . 104
Creating Mapping Statements. . . . 105
Mapping Statements Grid Interface. . . . 105
XPath Expressions. . . . 106
Predicates. . . . 107
XPath Expression Editor. . . . 110
Data Processor Functions. . . . 111
XPath Expressions Example. . . . 112
Creating An Expression. . . . 113
XMap Variables. . . . 113
Creating a Variable in the XMap Editor. . . . 113
XMap Example. . . . 113
XML Input Schema Example. . . . 114
XML Output Schema Example. . . . 115
XML Input Data. . . . 115
Input and Output XML Hierarchies. . . . 116
Mapping Statements in the Example. . . . 117
Group Statements Example. . . . 118

Chapter 7: Libraries. . . . 120

Libraries Overview. . . . 120
Library Structure. . . . 121
Element Properties. . . . 121
Library Management. . . . 121

Edit Libraries with the Library Editor. 122
Adding an Element with the Library Editor. 123
Editing the Element Properties with the Library Editor. 123
Testing a Library. 123
Generating the Library Objects. 124
Discarding the Library Objects. 124
Edit Libraries with the IntelliScript Editor. 124

Chapter 8: Schema Object. 125

Schema Object Overview. 125
Schema Object Overview View. 125
Schema Files. 126
Schema Object Schema View. 126
Namespace Properties. 127
Element Properties. 128
Simple Type Properties. 129
Complex Type Properties 130
Attribute Properties. 131
Schema Object Advanced View. 132
Creating a Schema Object. 132
Schema Updates. 133
Schema Synchronization. 134
Schema File Edits. 134

Chapter 9: Command Line Interface. 137

Command Line Interface Overview. 137
CM_console. 137

Chapter 10: Scripts. 140

Scripts Overview. 140
Script Components. 141
Component Types. 141
Component Names. 142
Adding a Global Component. 142
Adding a Local Component. 142
Script Component Properties. 143
Simple Properties. 143
Advanced Properties. 143
Component Property Values. 144
Script Startup Components. 144
Setting the Startup Component with the IntelliScript Editor. 144
Example Sources. 145
Example Source Highlighting. 145

Setting an Example Source in the IntelliScript Editor. . . . 145
Viewing an Example Source. . . . 146
IntelliScript Editor. . . . 146
Validate a Script. . . . 147
Sample Scripts. . . . 147
Importing a Sample Script. . . . 148

Chapter 11: Parsers. . . . 149

Parsers Overview. . . . 149
Platform-Independent Parsers. . . . 149
Newline Markers. . . . 149
File Paths. . . . 149
Parser Component Reference. . . . 150
Parser. . . . 150

Chapter 12: Script Ports. . . . 153

Script Ports Overview. . . . 153
Script Port Component Reference. . . . 153
AdditionalInputPort. . . . 153
AdditionalOutputPort. . . . 155
DocList. . . . 158
FileSearch. . . . 158
InputPort. . . . 159
LocalFile. . . . 159
OutputPort. . . . 159
Text. . . . 159
URL. . . . 160

Chapter 13: Document Processors. . . . 161

Document Processors Overview. . . . 161
Defining a Document Processor. . . . 161
Display of Document Processor Output. . . . 162
Document Processor Component Reference. . . . 162
AFPToXML. . . . 162
AsnToXml. . . . 162
ExcelToDataXml. . . . 163
ExcelToXml. . . . 163
ExcelToXml_03_07_10. . . . 164
ExpandFrameSet. . . . 165
ExternalJavaPreProcessor. . . . 165
ExternalPreProcessor. . . . 165
HIPAAValidator. . . . 165
PdfFormToXml_1_00. . . . 166

PdfToTxt_3_02. 166
PdfToTxt_4. 167
PowerpointToTextML. 167
ProcessByTransformers. 168
ProcessorPipeline. 168
RtfToTextML. 168
WordToXml. 168
XmlToDocument_372. 168
XmlToDocument_45. 169
XmlToExcel. 170
XmlToXlsx. 170
TextML XML Schema. 171
PdfToTxt_4 Table Configuration Editor. 172
Editor Options. 173
PDF Conversion Example. 174

Chapter 14: Formats. 176

Formats Overview. 176
Standard Format Properties. 177
Format Component Reference. 177
BinaryFormat. 178
CustomFormat. 179
HtmlFormat. 180
RtfFormat. 181
TextFormat. 181
XmlFormat. 182
Delimiters Component Reference. 183
CommaDelimited. 184
Delimiter. 184
DelimiterHierarchy. 185
EnclosingDelimiters. 185
HL7. 186
Positional. 186
PostScript. 187
RTF. 187
SGML. 187
SpaceDelimited. 187
TabDelimited. 187
Format Preprocessor Component Reference. 188
HtmlProcessor. 188
RtfProcessor. 188

Chapter 15: Data Holders. 189
Data Holders Overview. 189
XML Schemas. 189
Schema Encoding. 190
Included Schema Files. 190
Namespaces. 190
Mixed Content. 190
Unsupported Schema Features. 190
Precision of Numerical Data. 191
Using a Schema to Map Anchors. 192
IntelliScript Representation of Data Holders. 192
Mapping Mixed Content. 192
Mapping XSI Types. 192
Generating Valid XML. 193
Role of Schemas in Parsing. 193
Role of Schemas in Serialization and Mapping. 194
Variables. 194
Creating a User-Defined Variable. 195
System Variables. 195
Mapping Anchors to Variables. 197
Using Variables in Actions. 197
Initializing Variables at Runtime. 198
Variable Component Reference. 198
Variable. 198
Multiple-Occurrence Data Holders. 199
Attributes. 199
Indexing. 199
Destroying the Occurrences. 200

Chapter 16: Anchors. 201
Anchors Overview. 201
Marker and Content Anchors. 201
Other Anchor Types. 201
How Anchors and Delimiters Work Together. 202
Mapping Content Anchors to Data Holders. 202
Mapping to Variables. 203
Mapping to Multiple-Occurrence Data Holders. 203
Mapping to Mixed-Content Elements. 203
Defining Anchors. 203
Where to Define Anchors. 203
Sequence of Anchors. 204
Adding a Marker or Content Anchor. 204

Defining an Anchor. . . . 204
Standard Anchor Properties. . . . 205
How a Parser Searches for Anchors. . . . 206
Search Phases. . . . 206
Search Scope and Search Criteria. . . . 207
Adjusting the Search Phase. . . . 208
Adjusting the Search Scope. . . . 208
Adjusting the Search Criteria. . . . 210
Using Data Types to Narrow the Search Criteria. . . . 211
Anchors that Contain Nested Anchors. . . . 212
Anchor Component Reference. . . . 212
Alternatives. . . . 213
Content. . . . 215
DelimitedSections. . . . 218
EmbeddedParser. . . . 221
EnclosedGroup. . . . 222
ExtractRecord. . . . 224
FindReplaceAnchor. . . . 225
Group. . . . 227
Marker. . . . 230
RepeatingGroup. . . . 232
StructureDefinition. . . . 236
Searcher Component Reference. . . . 240
AttributeSearch. . . . 240
LearnByExample. . . . 241
NewlineSearch. . . . 241
OffsetSearch. . . . 241
PatternSearch. . . . 242
SegmentSearch. . . . 242
TextSearch. . . . 243
TypeSearch. . . . 244
Anchor Subcomponent Reference. . . . 244
AllStructure. . . . 244
AllStructureLocal. . . . 245
ChoiceStructure. . . . 245
ChoiceStructureLocal. . . . 246
Connect. . . . 247
EmbeddedStructure. . . . 247
RecordStructure. . . . 248
RecordStructureLocal. . . . 249
SequenceStructure. . . . 249
SequenceStructureLocal. . . . 250

Chapter 17: Transformers 252

Transformers Overview 252
Defining Transformers 252
- Using Transformers in Anchors 252
- Sequences of Transformers 253
- Default transformers 253
- Using Transformers as Document Processors 253
- Using Transformers in Serialization Anchors 254
- Using Transformers in Actions 254

Standard Transformer Properties 254
Transformer Component Reference 254
- AbsURL 255
- AddEmptyTagsTransformer 255
- AddString 256
- Base64Decode 256
- Base64Encode 257
- BidiConvert 257
- CDATADecode 258
- CDATAEncode 258
- ChangeCase 259
- CreateGuid 260
- CreateUUID 260
- DateFormatICU 260
- Dos96HebToAscii 263
- DynamicTable 263
- EbcdicToAscii 263
- EDIFACTValidation 263
- EncodeAsUrl 264
- Encoder 264
- ExternalTransformer 265
- FormatNumber 266
- FromFloat 267
- FromInteger 267
- FromPackDecimal 268
- FromSignedDecimal 268
- hebrewBidi 269
- HebrewDosToWindows 269
- HebrewEBCDICOldCodeToWindows 269
- hebUniToAscii 269
- hebUtf8ToAscii 269
- HtmlEntitiesToASCII 270

HtmlProcessor. 270
InjectFP. 270
InjectString. 271
InlineTable. 271
JavaTransformer. 272
LookupTransformer. 272
NormalizeClosingTags. 274
RegularExpression. 275
RemoveMarginSpace. 277
RemoveRtfFormatting. 277
RemoveTags. 277
Replace. 278
Resize. 279
ReverseTransformer. 279
RtfProcessor. 280
RtfToASCII. 280
SubString. 280
ToFloat. 281
ToInteger. 282
ToPackDecimal. 282
TransformationStartTime. 283
TransformByParser. 284
TransformByProcessor. 285
TransformByService. 285
TransformerPipeline. 286
XMLLookupTable. 287
XSLTTransformer. 287

Chapter 18: Actions. 289

Actions Overview. 289
How Actions Work. 289
Comparison Between Actions and Transformers. 290
Defining Actions. 290
Standard Action Properties. 290
Action Component Reference. 291
AddEventAction. 291
AggregateValues. 292
AppendListItems. 294
AppendValues. 295
CalculateValue. 296
CombineValues. 298
CreateList. 299
CustomLog. 300

DateAddICU. 301
DateDiffICU. 302
DownloadFileToDataHolder. 303
DumpValues. 304
EnsureCondition. 305
ExcludeItems. 308
Map. 308
Notify. 310
ResetVisitedPages. 310
RunMapper. 311
RunMapplet. 312
RunParser. 313
RunPCWebService. 315
RunSerializer. 315
RunXMap. 316
SetValue. 317
Sort. 318
ValidateValue. 319
WriteValue. 320
XSLTMap. 322
Action Subcomponent Reference. 323
OutputDataHolder. 323
OutputFile. 323
ResultFile. 324
StandardErrorLog. 324

Chapter 19: Serializers. 325

Serializers Overview. 325
Controlling How the Create Serializer Command Works. 325
Troubleshooting an Auto-Generated Serializer. 327
Creating a Serializer by Editing the Script. 328
Creating a Serializer within a RunSerializer Action. 328
Serialization Anchors. 328
Example of Serialization Anchors. 329
Sequence of Serialization Anchors. 329
Standard Serializer Properties. 330
Serializer Component Reference. 330
Serializer. 330
Serialization Anchor Component Reference. 331
AlternativeSerializers. 332
ContentSerializer. 333
DelimitedSectionsSerializer. 334
EmbeddedSerializer. 336

GroupSerializer. 337
RepeatingGroupSerializer. 338
StringSerializer. 340

Chapter 20: Mappers. 342

Creating a Mapper. 342
Components Nested within a Mapper. 342
Mapper Example. 343
Source XML. 343
Output XML. 343
Mapper Configuration. 344
Standard Mapper Properties. 344
Mapper Component Reference. 345
Mapper. 345
Mapper Anchor Component Reference. 346
AlternativeMappings. 347
EmbeddedMapper. 348
GroupMapping. 349
RepeatingGroupMapping. 350

Chapter 21: Locators, Keys, and Indexing. 352

Overview of Locators, Keys, and Indexing. 352
Example of Locators. 353
Input and Output. 353
Incorrect Solution. 353
Correct Solution. 354
Example of Indexing by Key. 354
Input. 354
Output. 355
Outline of the Transformation Approach. 355
Mapper Configuration. 355
Use of Indexing. 357
Source and Target Properties. 357
Source Property. 358
Target Property. 361
Standard Locator and Key Properties. 363
Locator and Key Component Reference. 363
Key. 363
Locator. 365
LocatorByKey. 366
LocatorByOccurrence. 367

Chapter 22: Streamers. 368

Streamers Overview. 368
Text Streamers. 369
Segments. 369
Simple Segments. 369
Complex Segments. 369
Example. 370
Header Concatenation. 370
Output of a Streamer. 370
Using Markers and Variables in Streamers. 371
Creating a Streamer. 371
XML Streamers. 373
Standard Streamer Properties. 375
Streamer Component Reference. 375
ComplexSegment. 376
ComplexXmlSegment. 376
JsonStreamer. 377
MarkerStreamer. 377
SimpleSegment. 379
SimpleXmlSegment. 380
Streamer. 381
StreamerVariable. 382
XmlSegment. 382
XmlStreamer. 383
Streamer Subcomponent Reference. 384
AddHeaderModifier. 384
AddStringModifier. 385
DoNothingModifier. 386
WellFormedModifier. 386
WriteSegment. 387

Chapter 23: Validators, Notifications, and Failure Handling. 388

Overview of Validators, Notifiers, and Failure Handling. 388
Failure Handling. 389
Using the Optional Property to Handle Failures. 389
Writing a Failure Message to the User Log. 390
Validators. 392
Standard Validator Properties. 392
Validator Component Reference. 393
AlternativeValidators. 393
EDIFACTValidation. 394
Enumeration. 395

LengthEquals. 396
MaxLength. 397
MaxNumber. 398
MinLength. 399
MinNumber. 400
NumberEquals. 401
ValidateByExpression. 402
ValidateByPattern. 403
ValidateByTransformer. 404
ValidateByType. 405
ValidateDate. 406
ValidatorPipeline. 407
Notifications. 408
Notification Component Reference. 409
Notification. 409
NotificationGroup. 409
NotificationHandler. 409
NotifyFailure. 410

Chapter 24: Validation Rules. 412

Validation Rules Overview. 412
Validation Rules Element Reference. 413
Assert Element Attributes. 413
List Element Attributes. 414
Lookup Element Attributes. 414
Rule Element Attributes. 415
Trace Element Attributes. 415
Variable Element Attributes. 415
XPath Editor. 416
XPath Extensions. 416
Edit the Validation Rules in an External Editor. 419
Create a Validation Rules Object. 419
Import a Data Transformation Service with Validation Rules. 419

Chapter 25: Custom Script Components. 421

Custom Script Components Overview. 421
Custom Component Example. 421
Custom Component Properties. 422
Developing a Custom Component in Java. 422
Java Interface Example. 423
Sample Custom Java Components. 423
Developing a Custom Component in C or C++. 424
C/C++ Property Value Length Limitation. 425

C/C++ Interface Example. 425
Sample Custom C/C++ Components. 425
Configuring a Custom Component. 426
Sample Scripts Containing Custom Components. 427

Index. 428

Preface

The *PowerCenter Express Data Transformation User 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.

Informatica Resources

Informatica Network

Informatica Network hosts Informatica Global Customer Support, the Informatica Knowledge Base, and other product resources. To access Informatica Network, visit https://network.informatica.com.

As a member, you can:

- Access all of your Informatica resources in one place.
- Search the Knowledge Base for product resources, including documentation, FAQs, and best practices.
- View product availability information.
- Review your support cases.
- Find your local Informatica User Group Network and collaborate with your peers.

Informatica Knowledge Base

Use the Informatica Knowledge Base to search Informatica Network for product resources such as documentation, how-to articles, best practices, and PAMs.

To access the Knowledge Base, visit https://kb.informatica.com. If you have questions, comments, or ideas about the Knowledge Base, contact the Informatica Knowledge Base team at KB_Feedback@informatica.com.

Informatica Documentation

To get the latest documentation for your product, browse the Informatica Knowledge Base at https://kb.informatica.com/_layouts/ProductDocumentation/Page/ProductDocumentSearch.aspx.

If you have questions, comments, or ideas about this documentation, contact the Informatica Documentation team through email at infa_documentation@informatica.com.

Informatica Product Availability Matrixes

Product Availability Matrixes (PAMs) indicate the versions of operating systems, databases, and other types of data sources and targets that a product release supports. If you are an Informatica Network member, you can access PAMs at https://network.informatica.com/community/informatica-network/product-availability-matrices.

Informatica Velocity

Informatica Velocity is a collection of tips and best practices developed by Informatica Professional Services. Developed from the real-world experience of hundreds of data management projects, Informatica Velocity represents the collective knowledge of our consultants who have worked with organizations from around the world to plan, develop, deploy, and maintain successful data management solutions.

If you are an Informatica Network member, you can access Informatica Velocity resources at http://velocity.informatica.com.

If you have questions, comments, or ideas about Informatica Velocity, contact Informatica Professional Services at ips@informatica.com.

Informatica Marketplace

The Informatica Marketplace is a forum where you can find solutions that augment, extend, or enhance your Informatica implementations. By leveraging any of the hundreds of solutions from Informatica developers and partners, you can improve your productivity and speed up time to implementation on your projects. You can access Informatica Marketplace at https://marketplace.informatica.com.

Informatica Global Customer Support

You can contact a Global Support Center by telephone or through Online Support on Informatica Network.

To find your local Informatica Global Customer Support telephone number, visit the Informatica website at the following link: http://www.informatica.com/us/services-and-training/support-services/global-support-centers.

If you are an Informatica Network member, you can use Online Support at http://network.informatica.com.

CHAPTER 1

Introduction to Data Transformation

This chapter includes the following topics:

- Data Transformation Overview, 22
- Data Transformation Process Architecture, 23
- Data Transformation Components, 24

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 a Data Transformation Script, XMap, Library, or Validation Rules object 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. The repository directory name is ServiceDB.

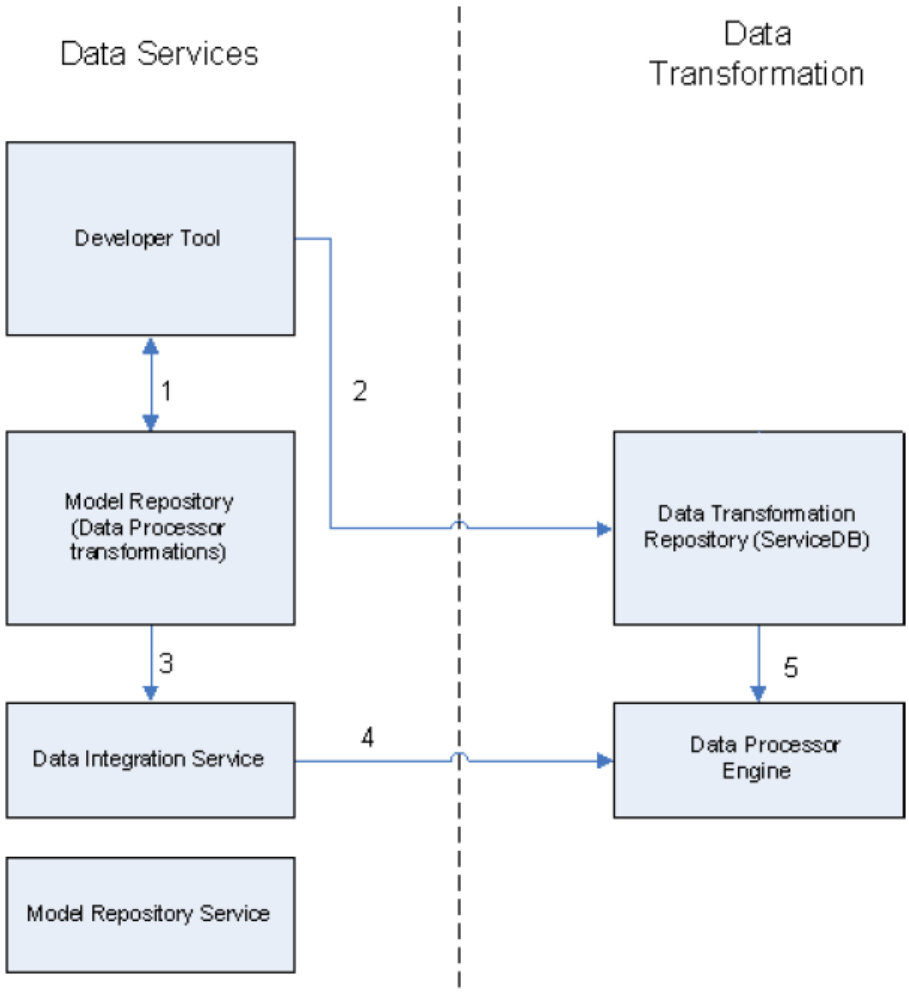
Data Processor Engine

A processor that runs objects in the Data Processor transformation or services that you create in Developer tool.

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 Scripts, Libraries 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 deploy an application that contains a Data Processor transformation to a Data Integration Service.
4. The Data Integration Service runs the application and calls the Data Processor Engine to process the transformation logic.

5. 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:

Library

Transform an industry message type input into other formats.

Mapper

Converts an XML source document to another XML document.

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.

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.

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 standalone component.

XMap

Converts a hierarchical source to another hierarchical structure. XMap has the same functionality as Mapper, but you can use a grid in the Developer tool to define the mapping.

CHAPTER 2

Data Processor Transformation

This chapter includes the following topics:

- Data Processor Transformation Overview, 25
- Data Processor Transformation Views, 26
- Data Processor Transformation Ports, 27
- Startup Component, 29
- References, 29
- Data Processor Transformation Settings, 30
- Events, 37
- Logs, 38
- Data Processor Transformation Development, 40
- Data Processor Transformation Import and Export, 44

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, JSON, and PDF documents. You can also convert structured formats such as ACORD, HIPAA, HL7, EDI-X12, EDIFACT, AFP, and SWIFT.

A mapping uses a Data Processor transformation to change documents from one format to another. The Data Processor transformation processes files of any format in a mapping. When you create a Data Processor transformation, you define components that convert the data.

A Data Processor transformation can contain multiple components to process data. Each component might contain other components.

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 an XMap, Script, or Library. An XMap converts an input hierarchical file into an output hierarchical file of another structure. A Library converts an industry messaging type into an XML document with a hierarchy structure or from XML to an industry standard format. A Script can parse source documents to hierarchical format, convert hierarchical format to other file formats, or map a hierarchical document to another hierarchical format.

Define Scripts in the Data Processor transformation IntelliScript 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 convert 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.

For more information, see the *Data Transformation User Guide*.

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** > **Show View** > **Other** > **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, XMap and Library 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 from a complex file reader. A Data Processor transformation can read input from a file, a buffer, or a streamed buffer from a complex file reader. 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 transformaton.

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 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.

Pass-Through Ports

You can configure pass-through ports to any 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.

To add a pass-through port, drag a port from another transformation in the mapping. You can also add ports in the **Ports** tab of the **Properties** view. Click **New** to add a pass-through port.

Note: When you add pass-through ports to a Data Processor transformation with relational input and hierarchical output, add the ports to the root group of the relational structure.

Data Processor transformations can include pass-through ports with custom data types.

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, Library, or a Script component as the startup component. In terms of Scripts, you can select one of the following types of components:

- Parser. Converts source documents to 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.
- Streamer. Splits large input documents, such as multi-gigabyte data streams, into segments.
- 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 or Library, you can also configure the startup component in a Script instead of in the **Overview** view.

References

You can define transformation references, such as schema or mapplet references, by selecting a schema or mapplet to serve as a reference. Some Data Processor transformations require a hierarchical schema to define the input or output hierarchy for relevant components in the transformation. To use the schema in the transformation, you define a schema reference for the transformation. You can also use a specialized action

named the RunMapplet action to call a mapplet from a Data Processor transformation. To call a mapplet, you must first define a mapplet reference for the transformation.

You can define transformation references, such as schema or mapplet references, in the transformation **References** view.

Schema References

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 must match the input encoding for Serializer or Mapper objects. The schema encoding must match the output encoding for Parser or Mapper 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.

Mapplet References

You can call a mapplet from a Data Processor transformation with the RunMapplet action. Before you add the RunMapplet action to a Data Processor transformation component, you must first define a reference to the mapplet that you want to call.

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:

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

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:

Setting	Description
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 transfomation 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.

Setting	Description
XML Special Characters Encoding	Determines the representation of XML special characters. You can select **None** or **XML.** - **None.** Leave as & < > " ' Entity references for XML special characters are interpreted as text, for example, the character > appears as `>` Default in None. - **XML**. Convert to & < > " ' Entity references for XML special characters are interpreted as regular characters. For example, > 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:

Setting	Description
Use Working Encoding	The output encoding is the same as the working encoding.
Other	The user selects the output encoding from the list.
XML Special Characters Encoding	Determines the representation of XML special characters. You can select **None** or **XML.** - **None.** Leave as & < > " ' Entity references for XML special characters are interpreted as text, for example, the character > appears as `>` Default. - **XML**. Convert to & < > " ' Entity references for XML special characters are interpreted as regular characters. For example, > appears as the following character: >
Same as Input Encoding	The output encoding is the same as the input encoding.
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.

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:

Setting	Description
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:

Setting	Description
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:

Setting	Description
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 disable it.

The following table describes the settings in the **Binary output port collection mode** area. You can select one of these options for binary output for a relational to hierarchical transformation with XML, Avro, or Parquet output, or for a Data Processor transformation Parser with Avro or Parquet output.

Setting	Description
Collect input rows to a single output	Determines whether the Data Processor transformation accumulates the relational input to a single binary output port.
Split output when size exceeds	Determines whether the Data Processor transformation divides the output into chunks based on a maximum stated output size.
Output row for each row (do not collect)	Determines whether the Data Processor transformation passes the output in separate rows.

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:

Setting	Description
Treat as xs:string	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.
Treat as xs:anyType	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.

The following table describes a processing setting that affects Streamer processing:

Setting	Description
Streamer chunk size	This setting defines the amount of data that the Streamer reads each time from an input file stream. The Data Processor transformation applies this setting to a Streamer with a file input.

The following table describes a processing setting that affects hierarchical to relational transformation processing:

Setting	Description
Enforce strict validation	This setting determines if the Data Processor transformation performs strict validation for hierarchical input. When strict validation applies, the hierarchical input file must conform strictly to its schema. This option can be applied when the Data Processor mode is set to **Output Mapping**, which creates output ports for relational output. This option does not apply to mappings with JSON input from versions previous to version 10.2.1.
Normalize XML input	This setting determines if the Data Processor transformation normalizes XML input. By default, the transformation performs normalization for XML input. In some cases, you might choose to skip automatic normalization to increase performance.

XMap Settings

The XMap setting defines how the Data Processor transformation processes XMap input elements that are not transformed to output elements. The unread elements are passed to a dedicated port named **XMap_Unread_Input_Values**. The setting takes affect only when the XMap is selected as the start-up component. The setting does not affect elements that the XMap processes.

To pass unread XMap elements to a dedicated port, enable the setting **Write unread elements to an additional output port**.

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:

Setting	Description
Schema location	Defines the schemaLocation for the root element of the main output document.
No namespace schema location	Defines the xsi:noNamespaceSchemaLocation attribute of the root element of the main output document.

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:

Setting	Description
As is	Do not add or remove empty elements. Default is enabled.
Full	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.
Compact	Removes empty elements from the output. If **Add for Elements** is enabled, then the Data Processor transformation removes only the optional elements. If **Add for Elements** is disabled, the Data Processor transformation removes all empty elements. The XML output might not be valid.

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

Setting	Description
Add for elements	When the output schema defines a default value for a required element, the output includes the element with a default value. Default is enabled.
Add for attributes	When the output schema defines a default value for a required attribute, the output includes the attribute with its default value. Default is enabled.
Validate added values	Determines whether the Data Processor transformation validates empty elements that are added by the Full mode output. Default is disabled. If **Validate added values** is enabled and the schema does not allow empty elements, the XML output might not be valid.

The following table describes the XML generation settings in the **Processing Instructions** area:

Setting	Description
Add XML processing instructions	Defines the character encoding and XML version of the output document. Default is selected.
XML version	Defines the XML version. The XML version setting has the following options: - 1.0 - 1.1 Default is 1.0.
Encoding	Defines the character encoding that is specified in the processing instruction. The Encoding setting has the following options: - 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. - Custom. Defines the output encoding in the processing instruction. The user types the value in the field.
Add custom processing instructions	Adds other processing instructions to the output document. Enter the processing instruction exactly as it appears in the output document. Default is Disabled.

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

Setting	Description
Add XML root element	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.
Root element name	Defines a name for the root element to add to the output document.

Events

An event is a record of a processing step from a component in the Data Processor transformation. In a Script or Library, 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:

Event Type	Description
Notification	Normal operation.
Warning	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.
Failure	The Data Processor transformation ran, but a component failed. For example, a required input element was empty.
Optional Failure	The Data Processor transformation ran, but an optional component failed. For example, an optional anchor was missing from the source document.
Fatal Error	The Data Processor transformation failed because of a serious error. For example, the input document did not exist.

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, Library 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

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_Dirctory>/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:

```
home/<Installation_Dirctory>/DataTransformation/UserLogs
```

Data Processor Transformation Development

Use the New Transformation wizard to auto-generate a Data Processor transformation, or create a blank Data Processor transformation and configure it later. If you create a blank Data Processor transformation, you must select to create a Script, XMap, Library, or Validation Rules object in the transformation. A Script can parse source documents to hierarchical format, convert hierarchical format to other file formats, or map a hierarchical document to another hierarchical format. An XMap converts an input hierarchical file into an output hierarchical file of another structure. A Library converts an industry messaging type into an XML document with a hierarchy structure or from XML to an industry standard format. Choose the schemas that define the input or output hierarchies.

1. Create the transformation in the Developer tool.
2. For a blank Data Processor transformation, perform the following additional steps:
   a. Add schema references that define the input or output XML hierarchies.
   b. Create a Script, XMap, Library, or Validation Rules object.
3. Configure the input and output ports.
4. Test the transformation.

Create the Data Processor Transformation

Create a Data Processor transformation in the Developer tool. If you create a blank Data Processor transformation, you must then create a Script, XMap, Library, or Validation Rules object in the transformation. Alternatively, you can use the New Transformation wizard to auto-generate a Data Processor transformation.

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. Select whether to create Data Processor transformation with a wizard or to create a blank Data Processor transformation.
5. If you selected to create a blank Data Processor transformation, click **Finish**.

   The Developer tool creates the empty transformation in the repository. The **Overview** view appears in the Developer tool.

6. If you selected to create an Data Processor transformation with a wizard, perform the following steps:
   a. Click **Next**.
   b. Select an input format.
   c. Browse to select a schema, copybook, example file, or specification file if required for certain input formats such as COBOL, JSON, or ASN.1.
   d. Select an output format.
   e. Browse to select a schema, copybook, example file, or specification file if required for the output format.
   f. Click **Finish**. The wizard creates the transformation in the repository.

   The transformation might contain a Parser, Serializer, Mapper, or an object with common components. If you selected a schema, copybook, example file, or specification file the wizard also creates a schema in the repository that is equivalent to the hierarchy in the file.

Select the Schema Objects

Choose the schema objects that define the input or output 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 **Refererences** 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 a hierarchical schema file.
4. Click Finish to add the schema reference to the Data Processor transformation.

Create Objects in a Blank Data Processor Transformation

Create a Script, Library, XMap, or Validation Rules 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 schema reference and example source file.

1. In the Data Processor transformation **Objects** view, click **New.**
2. Enter a name for the Script 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 uses the text to test 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 hierarchal documents. You select the element in the schema that is the root element for the input hierarchy.

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.

   To import a new Schema object, click **Create a new schema object**.
6. If you have a sample hierarchical 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 hierarchical 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 hierarchal file. 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.

Creating a Library

Create a Library object on the Data Transformation **Objects** view. Select the message type, component and name. Optionally, you can define a sample message type source file that you can use to test the Library object.

Before you create a Library in the Data Processor transformation, install the library software package on your computer.

1. In the Data Processor transformation **Objects** view, click **New.**

2. Select Library and click **Next.**
3. Browse to select the message type.
4. Choose to create a Parser or Serializer.

   Create a Parser if the Library object input is a message type and the output is XML. Create a Serializer if the Library object input is XML and the output is a message type.
5. If the Library is the first component to process data in the Data Processor transformation, enable **Set as startup component**.

   Click **Next.**
6. If you have a sample message type source file that you can use to test the Library with, browse for and select the file from the file system.

   You can change the sample file.
7. Click **Finish**.

   The Developer tool creates a view for each message type that you create. Click the view to access the mapping.

Creating Validation Rules

Create a Validation Rules object in the Data Processor transformation **Objects** view.

1. In the Data Processor transformation **Objects** view, click **New.**
2. Select Validation Rules and click **Next.**
3. Enter a name for the Validation Rules.
4. If you have a sample XML file that you can use to test the Validation Rules with, browse for and select the file from the file system.

   You can change the sample XML file.
5. Click **Finish**.

   The Developer tool creates a Validation Rules object and opens it in the Validation Rules editor.

Adding an Example Source

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

You can add an example source when you create a Script, XMap, Library, or Validation Rules. Once selected, the example source is added to the Model repository. Due to Model repository limitations, the exaple source file size is limited to 5 MB.

You can change the example source.

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.

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 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 Multiple Data Transformation Services

You can import a directory of Data Transformation services from the machine where you saved the directory. When you import Data Transformation services to the Developer Model repository, the Developer tool imports the transformations, schemas and example data with the . cmw files. If you need to import many services, import a directory of services instead of one service at a time.

1. Click **File** > **Import**,

   The **Import** dialog box appears.

2. Select **InformaticaImport Data Transformation Services (Folder)** and click **Next**.

   The **Import Data Transformation Service** page appears.

3. Browse to the directory that you want to import.

4. Browse to a location in the Repository where you want to save the transformations, then click **Finish**.

   The Developer tool imports the transformations, schemas and example data with the `.cmw` file.

Importing a Data Transformation Service

You can import a Data Transformation service .cmw file to the Model repository to create a Data Processor transformation. The Developer tool imports the transformation, schemas and example data with the .cmw file.

1. Click **File** > **Import**,

   The **Import** dialog box appears.

2. Select **Informatica Import Data Transformation Service (Single)** and click **Next**.

   The **Import Data Transformation Service** page appears.

3. Browse to the service .cmw file that you want to import.

   The Developer tool names the transformation according to the service file name. You can change the name.

4. Browse to a location in the Repository where you want to save the transformation, then click **Finish**.

   The Developer tool imports the transformation, schemas and example data with the `.cmw` file.

5. To edit the transformation, double-click the transformation in the **Object Explorer** view.

Exporting a Mapping with a Data Processor Transformation to PowerCenter

When you export a mapping with a Data Processor transformation to PowerCenter, you can export the objects to a local file, and then import the mapping into PowerCenter. Alternatively, you can export the mapping directly into the PowerCenter repository.

1. In the **Object Explorer** view, select the mapping to export. Right-click and select **Export**.

   The **Export** dialog box appears.
2. Select **Informatica** > **PowerCenter**.
3. Click **Next**.

   The **Export to PowerCenter** dialog box appears.
4. Select the project.
5. Select the PowerCenter release.
6. Choose the export location, a PowerCenter import XML file or a PowerCenter repository.
7. Specify the export options.
8. Click **Next**.

   The Developer tool prompts you to select the objects to export.
9. Select the objects to export and click **Finish**.

   The Developer tool exports the objects to the location you selected. If you exported the mapping to a location, the Developer tool also exports the Data Processor transformations in the mapping, as services, to a folder at the location that you specified.
10. If you exported the mapping to a PowerCenter repository, the services are exported to the following directory path: `%temp%\DTServiceExport2PC\`

    The export function creates a separate folder for each service with the following name: `<date><serviceFullName>`

    If the transformation includes relational mapping, a folder is created for relational to hierarchical mapping, and a separate folder for hierarchical to relational mapping.
11. Copy the folder or folders with Data Processor transformation services from the local location where you exported the files to the PowerCenter ServiceDB folder.
12. If you exported the mapping to a PowerCenter import XML file, import the mapping to PowerCenter. For more information about how to import an object to PowerCenter, see the *PowerCenter 9.6.0 Repository Guide*.

CHAPTER 3

Wizard Input and Output Formats

This chapter includes the following topics:

- Wizard Input and Output Formats Overview, 47
- ASN.1 Library, 47
- Avro, 56
- COBOL Processing Library, 61
- JSON, 64
- Parquet, 67
- XML, 69

Wizard Input and Output Formats Overview

You can use a wizard to create an auto-generated Data Processor transformation with input and output formats such as COBOL, XML, ASN.1, relational, or JSON. You can also use the wizard to transform user-defined formats.

Create a Data Processor transformation and select the input and output formats through the Data Processor transformation wizard. Select from existing formats or create user defined formats. For certain formats, such as ASN.1, XML, JSON, or COBOL, add a schema, specification file, example file or copybook that defines the expected structure for the input or output.

The wizard creates a transformation with relevant Script, XMap, or Library objects that serve as templates to transform the input format to the output format. The Data Processor transformation creates a transformation solution according to the formats selected and the specification file, the example file, or the copybook. The transformation might not be complete, but it contains components that you connect and customize to complete the transformation definition.

ASN.1 Library

The ASN.1 library processes data that complies with the Abstract Syntax Notation One (ASN.1) standard. The ISO and ITU-T organizations maintains the ASN.1 standard, a standard that the telecommunication industry uses. The standard uses a protocol specification syntax that represents data in compact structures, in binary or text formats.

Use the wizard to create a Data Processor transformation that parses ASN.1 protocol specifications. The Developer tool creates the following objects:

- An XML schema used to represent the ASN.1 data.
- A Parser, Serializer, and Mapper that contain an `AsnToXml` document processor. The processor input data conforms to the protocol specification. The output represents the same data in XML.

The transformation contains objects that you configure to process ASN.1 source data. The current ASN.1 library release processes input in the ASN.1 BER encoding.

The library includes a copy of the OSS Nokalva ASN-1Step utility. Use the utility to edit ASN.1 files. You can access the ASN.1 library services through the C/C++ API or the .NET API.

Note: The ASN.1 library transformations do not validate their input. If the input is not valid, the transformations might fail or might generate unexpected output.

ASN.1 Protocol Specification

When you create a Data Processor transformation for the ASN.1 library, you select an ASN.1 protocol specification for the transformation. The specification defines the expected input structure.

An ASN.1 protocol specification is a schema for an ASN.1 data structure. The specification is a text file with an `*.asn` extension.

The protocol specification defines the expected structure for input data. The following lines from a protocol specification define a nested data structure:

```
MobileTerminatedCall ::= [APPLICATION 10] SEQUENCE
{
    basicCallInformation        MtBasicCallInformation    OPTIONAL,
    locationInformation         LocationInformation       OPTIONAL,
    equipmentInformation        EquipmentInformation      OPTIONAL,
    basicServiceUsedList        BasicServiceUsedList      OPTIONAL,
    supplServiceUsedList        SupplServiceUsedList      OPTIONAL,
    camelServiceUsed            CamelServiceUsed          OPTIONAL,
    valueAddedServiceUsedList   ValueAddedServiceUsedList OPTIONAL,
    dualServiceRequested        DualServiceCode           OPTIONAL,
    operatorSpecInformation     OperatorSpecInfoList      OPTIONAL
}

...


ChargeableSubscriber ::= CHOICE
{
    simChargeableSubscriber SimChargeableSubscriber,
    minChargeableSubscriber MinChargeableSubscriber
}

...

SimChargeableSubscriber ::= [APPLICATION 199] SEQUENCE
{
    imsi     Imsi   OPTIONAL,
    msisdn   Msisdn OPTIONAL
}
```

The entire structure is a protocol data unit (PDU). A single protocol specification can define multiple PDUs. A protocol specification defines multiple types of data structures with different top-level elements.

ASN.1 Input Message Structure

The ASN.1 input is in binary format. The wizard creates a transformation for ASN.1 that includes the `AsnToXml` document processor. The document processor shows the input in an XML representation.

An ASN.1 source usually contains many messages. Each message is a data structure that conforms to the protocol specification. The source can contain separator strings between the messages, and there can be a header string before the first message.

The data is in a binary format. The following example contains data in the BER encoding:

```
_nSOHTe>□SSUB□XSYN_TSOHSOH_YSTX13_WBEL1000000_BELETXUSD□PDC1□jSO_i
SOHSOH_USSOHETX_hSTXENOÅ_tSOHENOfu□jDC2□iSO_hSOHSOH_gENO-0400□<<□7
SUB_8SOHNUL_:SOHSOH_ETBFF595991799503□7SUB_8SOHSOH_:SOHSOH_ETBFF5
95993269503_GSSOHSOH□ySYN□vDC2□NACK_METX595□□ACK_~ETX002gM□n!_mSOHSOH
_pSTXVD_oDC2Voice Mail  Retrie□n$_mSOHSTX_pSTXVR_oNAKVoice Mail
Retrievalcþjû□EM>□GSYN_SOHBSE@BNULX…E__CANACK…)EOTidUS□)EOT_!SOHSI□
,NAKPSO20040130160701_hSOHSOH__SOH[□
DC3□FSSI_8SOHNUL_BSSOHe_;STX+ETX□fENQ_DC4SOHSTX□&-□''□$US_ZSTX11_3
SOHNUL_4SOHSOH_JSOHSOH_KSOHSOH_dSOHSOH□Fm□Ej_BSOHD_iSOHSOH□,STXSI_
,ETXSOHSOH_□SOHNUL_,NULSOHSOH□@:□?SUB_GSTX00_>SOHNUL_ASOH[_DSOHx_U
SOHN_\SOHI□?SUB_GSTX01_>SOHNUL_ASOH[_DSOHx_USOHN_\SOHI□VSO□U
_TSOHSOH_f
```

The `AsnToXml` document processor converts the data to the ASN.1 XER encoding, which is an XML representation. When the Data Processor transformation wizard creates a transformation for ASN.1, it includes the `AsnToXml` document processor in the transformation.

The following example contains the XML that the `AsnToXml` document processor generates:

```
<mobileTerminatedCall>
  <basicCallInformation>
    <chargeableSubscriber>
      <simChargeableSubscriber>
        <imsi>45404200</imsi>
        <msisdn>85290469</msisdn>
      </simChargeableSubscriber>
    </chargeableSubscriber>
```

The XML hierarchy corresponds to the input message structure.

Installing the ASN.1 Library

To create transformations that process ASN.1 data, install the library on the development environment. To run the transformations as services, install the library on the production environment. You must obtain and configure an individual license for each host machine on which you install the library.

1. For each development environment, perform the following actions:
   a. Run the `hostinf` utility.

      The following message appears:

      ```
      HOST = "<host_name>/<code>"
      ```

   b. Copy the line that contains the host name and code.
2. Send an email to Informatica Global Customer Support. Include the following information:
   - The type of machine for each production environment

- The host name and code for each development environment

Informatica Global Customer Support sends you the `ossinfo` license files.

3. The step you take to install the ASN.1 Library depends on the environment. The following table describes the actions to take for each environment:

Environment	Actions
Development	1. Close the Developer tool. 2. Copy the installation file to the Data Transformation installation directory, and then unzip it.
Production	1. Copy the installation file to the Data Transformation installation directory. 2. On a Windows machine, unzip the installation file. On a Linux or UNIX machine, untar the installation file.

4. For each machine, save the associated `ossinfo` license file in the Data Transformation installation directory.

   For example, if you installed Data Transformation in the default location on a Windows platform, save the `ossinfo` file in the following directory:

   ```
   C:\Informatica\<version>\DataTransformation
   ```

Uninstalling the ASN.1 Library

On a Windows platform, uninstall the ASN.1 library before you uninstall Data Transformation. On a UNIX or Linux platform, the order of uninstallation is not important.

If you uninstall Data Transformation before the ASN.1 library on Windows, the uninstallation does not completely remove the library. To completely remove the library, do the following steps:

1. Restart Windows.
2. In the Windows Explorer, delete the Data Transformation installation directory.
3. Delete the following directory:

   ```
   C:\Program Files\Common Files\InstallShield\Universal\common\Gen1
   ```

Upgrading the ASN.1 Library

Before you install a new version of the ASN.1 library, you must uninstall the previous version.

Creating an ASN.1 Library Transformation

To create a Data Processor transformation for ASN.1 input, use the Data Processor transformation wizard. The wizard creates the transformation in the Model repository with a Parser, Serializer, Mapper, and an object with common components. Use the IntelliScript editor to edit these components.

1. Create the transformation with the Data Processor transformation wizard. Add an ASN.1 specification that defines the expect input structure.
2. Use the IntelliScript editor to edit and customize the Parser, Serializer, and Mapper that the wizard created.

Step 1. Create an ASN.1 Library Transformation with the Wizard

Create a Data Processor transformation with ASN.1 input. The Data Processor transformation wizard creates the transformation with components that transform ASN.1 input into XML.

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. Select **Create a data processor using a wizard** and click **Next**.
5. Select ASN.1 as the input format and click **Next**.
6. Browse to select an ASN.1 definition file. Click **Next**.

   Typically, the file has an `*.asn` extension.
7. Select an output format and click **Next**.
8. Click **Finish**.

   The wizard creates the transformation in the Model repository with a Parser, Serializer, Mapper, and an object with common components. The Parser, Serializer, and Mapper are configured with an AsnToXml document processor that converts the input to XML. The wizard also creates an XML schema in the repository that is equivalent to the protocol specification.

Step 2. Edit the ASN.1 Library Transformation

The wizard creates an ASN.1 Library transformation in the Model repository with a Parser, Serializer, Mapper, and an object with common components. You must edit the transformation components to add an example source, anchors, and actions.

The Parser, Serializer, and Mapper are configured with an AsnToXml document processor that converts the input to XML. The AsnToXml document processor converts the input data to the ASN.1 XER encoding, which is an XML representation.

1. In the **Objects** view, double-click the Parser, Serializer, or Mapper to open it in the IntelliScript editor.

   The transformation contains only the AsnToXml document processor. It does not contain an example source, anchors, or actions.
2. Configure the properties of the AsnToXml processor as required.

   The properties are described in the following section.
3. Add an example source to the Parser, Serializer, or Mapper.

   The example source must contain one or more messages that conform to the protocol specification. In the current release, the example must be in the BER encoding (a `*.ber` file).
4. Display the example source.

   It appears as an XML representation of the input data.
5. Under the `contains` line, add components such as anchors and actions that process the XML and generate output.

Installing the ASN-1Step Utility

Use the ASN-1Step development environment to edit and validate ASN.1 files. You can install ASN-1Step on a Windows computer. You can use ASN-1Step to edit ASN.1 protocol specifications and source data.

1. Run the ASN.1Step setup file.

   The setup file has a name such as `asnstep_winVVV.exe`, where `VVV` is a version number.
2. At the prompt, select the option to **Specify Location of ossinfo Manually**.
3. Browse to the Data Transformation installation directory where you stored the `ossinfo` license file.
4. Follow the on-screen instructions to complete the installation.

   For more information about using ASN-1Step, see the online help included in the utility.

ASN.1 Streamer

An `AsnStreamer` component converts and splits a large multi-message ASN.1 document into ASN.1 messages. The `AsnStreamer` can pass each message to an appropriate transformation.

ASN.1 Streamer Properties

The **AsnStreamer** component . splits and converts a large multi-message ASN.1 document into ASN.1 messages.

Each ASN.1 message is converted into XML.

The **AsnStreamer** must be defined at the global level of the Script and it must be the startup component of the transformation.

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

Property	Description
asn_file	Defines an ASN.1 specification file.
count	Defines the maximum number of converted ASN.1 messages to pass to the transformation.
encoding	Defines the ASN.1 encoding.
header	Defines a header to exclude from the ASN.1 file. The **header** property has the following options: - NewlineSearch. The header is a newline. - OffsetSearch. Define the header according to the number of characters from the beginning of the file. - PatternSearch. Define the header with a regular expression. - TextSearch. Define the header by an explicit string or a string that you retrieve dynamically from the source document.
max_lookup_size	Defines the maximum quantity of new data, in kilobytes, that the **AsnStreamer** searches for each new segment. For optimal performance, set this property to twice the maximum possible segment size. When an application activates a deployed **AsnStreamer** 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.

Property	Description
no_constraints	Determines whether to process the ASN file with constraints. The **no_constraints** property has the following options: - true. Process the ASN file without constraints. - false. Process the ASN file with constraints. Default is false.
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.
pdu_type	Defines the PDU type. Use this property to clarify an ambiguity.
separator	Defines text to ignore between records. The **separator** property has the following options: - NewlineSearch. The separator is a newline. - OffsetSearch. Define the separator by the number of characters from the end of the previous record. - PatternSearch. Define the separator by a regular expression. - TextSearch. Define the separator by an explicit string or a string that you retrieve dynamically from the source document.
xml_root	Define an XML root element to contain the converted ASN.1 messages.

AsnToXml Document Processor

The `AsnToXml` document processor converts ASN.1 data to XML.

The processor input format must be the BER encoding format that the ASN.1 standard defines. The output XML is identical to the ASN.1 XER encoding.

Note: This processor is available when you install the Data Transformation ASN.1 library.

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

Property	Description
asn_file	Defines an ASN.1 specification file.
header	Defines a header to exclude from the ASN.1 file. The **header** property has the following options: - NewlineSearch. The header is a newline. - OffsetSearch. Define the header according to the number of characters from the beginning of the file. - PatternSearch. Define the header with a regular expression. - TextSearch. Define the header by an explicit string or a string that you retrieve dynamically from the source document.
no_constraints	Determines whether to process the ASN file with constraints. The **no_constraints** property has the following options: - true. Process the ASN file without constraints. - false. Process the ASN file with constraints. Default is false.
pdu_type	Defines the PDU type. Use this property to clarify an ambiguity.

Property	Description
process_first_message	Determines whether to process the entire CDR file. The **process_first_message** property has the following options: - true. Only process the first record. - false. Process the entire CDR file. Default is false.
separator	Defines text to ignore between records. The **separator** property has the following options: - NewlineSearch. The separator is a newline. - OffsetSearch. Define the separator by the number of characters from the end of the previous record. - PatternSearch. Define the separator by a regular expression. - TextSearch. Define the separator by an explicit string or a string that you retrieve dynamically from the source document.

Example

In an ASN.1 BER file, there is a header before the first message and separators between the subsequent messages. The header and the separators are random sequences of hexadecimal 00 and FF bytes, of any length. For example, a header or separator might have the following form:

```
00 00 FF 00 FF FF FF 00 FF 00
```

The following configuration enables the `AsnToXml` processor to find the header and separators. The `header` and `separator` properties use regular expressions to define the pattern.

```
pre_processor = AsnToXml
  asn_file = SGSN-CDR-def-v2009A.asn
  header = PatternSearch
    pattern = "[\x00|\xFF]+"
  separator = PatternSearch
    pattern = "[\x00|\xFF]+"
```

The processor ignores the headers and separators, and it processes the messages between them.

ASN.1 Transformers

Transformers modify the output of other components. You can use transformers in Scripts within components such as anchors, serialization anchors, and actions. You can use transformers as document processors.

The following table describes the general properties of the ASN.1 transformer:

Property	Description
disabled	Determines whether the transformation ignores the component and all its child components. Choose one of the following options: - Selected. The transformation ignores the component. - Cleared. The transformation applies the component. Use this property for testing, debugging, and making minor modifications in a project without deleting the existing component. Default is cleared.
name	A descriptive label for the component. This label appears in the `events.cme` file and the **Events** view. Use the **name** property to identify which component caused the event.

Property	Description
optional	Determines whether a failure in the component causes the parent component to fail. Choose one of the following options: - Selected. The parent component is unaffected when the component fails. - Cleared. The parent component fails when the component fails. Default is cleared.
remark	A user-defined comment that describes the purpose or action of the component.

Viewing Advanced Properties

By default, advanced properties are hidden in the script panel of the IntelliScript editor.

▶ To view advanced properties for a specific component, in the IntelliScript editor press the Right Arrow key or click the double right arrow to the right of the component.

BCDSwapToString

The **BCDSwapToString** transformer transforms packed binary code decimal input into a string. The transformer swaps the nibbles for each byte, truncates the input to the first byte containing FF, and converts the result into a decimal string.

Example

You provide binary input that is equivalent to the following input to a **BCDSwapToString** transformer:

```
35 54 FF 54 54
```

The transformer swaps nibbles in each byte. When the transformer reaches byte FF, the transformer stops processing input and the output is truncated.

The transformer provides the following output:

```
5345
```

BCDToString

The **BCDToString** transformer converts packed binary code decimal input into a string. A binary code decimal number is a number coded so each nibble represents a decimal digit. A nibble is a half byte. The output is in little endian format, where the order of bytes is reversed so that the last byte in the input is the first byte in the output.

Example

You provide binary input that is equivalent to the following input to a **BCDToString** transformer:

```
52 54 98
```

The transformer takes the input and reverses the order of the bytes in the output.

The transformer provides the following string output:

```
985452
```

HexSequenceToDecimal

The **HexSequenceToDecimal** transformer converts an even number of characters representing a series of hexadecimal numbers to a decimal number.

Example

You provide the following hexadecimal input to a **HexSequenceToDecimal** transformer:

```
10170A
```

The transformer translates each input byte from hexadecimal to decimal and provides the following output:

```
162310
```

HexToDecimal

The **HexToDecimal** transformer converts an even number of characters representing a series of hexadecimal numbers to a series of decimal numbers.

The following table describes an additional property of the **HexSequenceToDecimal** transformer:

Property	Description
endian	Determines how the transformation stores data in memory. The **endian** property has the following options: - big_endian. The transformation stores the most significant byte in the smallest address. The bytes are in the same order as the input. - little_endian. The transformation stores the least significant byte in the smallest address. The bytes are in the reverse order of the input. This is the default setting.

Example

You provide the following hexadecimal input to a **HexToDecimal** transformer:

```
1017
```

The transformer translates the full input number from hexadecimal to decimal. By default, the transformer produces a little-endian transformation. Little-endian transformations reverse the order of the bytes.

The transformer provides the following output:

```
5904
```

Swap

The **Swap** transformer takes hexadecimal input in bytes and swaps odd and even nibbles in each byte. A nibble is a half byte.

Example

You provide the following hexadecimal input to a **Swap** transformer:

```
000102
```

The transformer swaps the nibbles in each byte, then provides the following hexadecimal output:

```
001020
```

Avro

Use the wizard to create a transformation with Avro input or output. When you create a Data Processor transformation to transform the Avro format, you select an Avro schema or example file that defines the

expected structure of the Avro data. The wizard creates components that transform Avro format to other formats, or from other formats to Avro format. These components can include a relational to hierarchical mapping, a hierarchical to relational mapping, and an XMap. After the wizard creates the transformation, you can further configure the transformation to determine the mapping logic.

Apache Avro is a data serialization system in binary or other data formats. Avro data is in a format that might not be directly human-readable. For more information about Avro, see http://avro.apache.org/

Note: Use binary encoded Avro to create a transformation with Avro input or output. Avro input or output in other formats is not processed.

A transformation that reads Avro input or output relies on a schema. When the transformation reads or writes Avro data, the transformation uses the schema to interpret the hierarchy.

When you select an example file to define the Avro hierarchy, the wizard also saves the first record in the file as a separate test file. You can use this file to test the transformation. To find the file, in the **Ports** panel of the **Overview** view, check the path file listed in the **Input Location** field.

When you create a Data Processor transformation that transforms Avro to hierarchical format, or hierarchical format to Avro, the wizard creates an XMap component in the transformation. The XMap editor shows the hierarchical schema nodes and the Avro schema nodes. Use the XMap editor to link the nodes and define the transformation logic. For more information about the XMap object and editor, see "XMap Overview" on page 88.

When you create a transformation that transforms Avro to relational format, or relational format to Avro, the wizard creates a relational mapping. The **Ports** panel in the **Overview** view shows the Avro hierarchical schema nodes and the relational ports. Use the **Ports** panel to link the hierarchical elements to the relational ports and groups. For more information about transforming relational data, see "Relational Input and Output Overview" on page 70.

After you create a Data Processor transformation for Avro input, you add it to a mapping with a complex file reader. The complex file reader passes Avro input to the transformation. For a Data Processor transformation with Avro output, you add a complex file writer to the mapping to receive the output from the transformation.

Avro Input and Complex File Reader

In order for Data Processor transformation to transform Avro input, the transformation receives data input from a complex file reader object. After you create and configure the transformation, you add the transformation to a mapping and connect the input port to the output port of the complex file reader.

The complex file reader provides input to a Streamer component that the Data Processor transformation wizard creates as part of the transformation. If the transformation transforms Avro input to a custom format or relational format, there is no need to change the Streamer settings. You specify a custom output format by selecting **Other** as the output format in the wizard.

If the transformation transforms Avro input to JSON, XML, or Avro format, the wizard creates an XMap to map the output format. You must identify the XMap to the Streamer, so that the transformation processes data correctly.

You must also configure the complex file reader to process Avro input. The output port for the complex file reader should be set to binary format. Similarly, the input port for the Data Processor transformation should be set to binary input.

Avro Data Compression with the Snappy Codec

You can compress Avro data with the complex file reader. If you use the Snappy codec for Avro data compression, you must update the Snappy codec .jar file before you test or run the transformation.

To use the Snappy codec, replace the default Snappy .jar file in the Informatica server installation and in the Hadoop environment with the updated version. The updated `snappy-java-1.0.4.1.jar` file is available at the following link: http://mvnrepository.com/artifact/org.xerial.snappy/snappy-java/1.1.0.1

Updating the Snappy Codec to Enable Avro Data Compression

To enable Avro data compression with the Snappy codec, replace the default Snappy .jar file with the updated version.

1. On the machine where you installed the Informatica server, replace the `snappy-java-1.0.4.1.jar` file in the server installation with the `snappy-java-1.1.0.1.jar` file. Replace the .jar at the following path: `<Server_Installation>\services\shared\hadoop\<Hadoop_Distribution>\lib`
2. On the machines where you installed and run Hadoop, replace the `snappy-java-1.0.4.1.jar` file with the `snappy-java-1.1.0.1.jar` file. Replace the .jar at the following path: `<Hadoop_rpm>\services\shared\hadoop\<Hadoop_Distribution>\lib`

Configure a Transformation with Avro Input

To create a Data Processor transformation for Avro input, use the Data Processor transformation wizard. The wizard creates the transformation in the Model repository with the components you need to transform Avro input. Use the IntelliScript editor to edit the Streamer, and the XMap editor to edit an XMap, if included in the transformation. Add the transformation to a mapping with a complex file reader.

1. Create the Data Processor transformation with the New Transformation wizard. Add an Avro schema or example file that defines the expect input structure.
2. If the transformation has XML, JSON, or another structured format as output, use the XMap editor to edit the XMap in the transformation.
3. Use the IntelliScript editor to edit and customize the Streamer in the transformation. If the transformation has XML, JSON, or another structured format output, edit the Streamer to identify the XMap in the transformation.
4. Add the Data Processor transformation to a mapping with a complex file reader. The transformation should remain set to binary input, the default setting for Avro input. Configure the complex file reader to process Avro. The output setting remains set to binary, the default setting. Link the complex file reader output port to the Data Processor transformation input port to provide the Avro input to the transformation.

Step 1. Create a Transformation that Transforms Avro

Create a Data Processor transformation with Avro input, Avro output, or both.

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. Select **Create a data processor using a wizard** and click **Next**.
5. Select Avro or another input format and click **Next**.

6. If you selected Avro as the input format, browse to select an .xsd schema file or sample Avro file. Click **Next**.

   Developer adds an .xsd schema file representing the Avro hierarchy to the Model Repository. If you select a sample file, Developer creates a test file from the first record in the sample file. You can use this file to test the transformation. To find the file, in the **Ports** panel of the **Overview** view, check the path file listed in the **Input Location** field.
7. Select Avro or another output format and click **Next**.
8. If you select Avro as an output format, browse to select a related schema or sample Avro file. Click **Next**.
9. Click **Finish**.

   The Developer tool creates the transformation in the repository with the relevant components, such as an XMap to transform the Avro hierarchy into another hierarchy format. The **Overview** view appears in the Developer tool.
10. To edit the components in the transformation, in the **Objects** view, double click the transformation component to open it in the relevant editor.

Step 2. Edit the XMap

To configure a Data Processor transformation XMap object, add mapping statements.

1. To open the XMap editor, on the Data Processor transformation **Objects** click the XMap object.
2. To create a Map, Group, or Repeating Group mapping statement, in the XMap editor drag and drop from a node in the input hierarchical schema to a node in the output hierarchical schema.

   The XMap editor creates a map link between the nodes. The mapping statement appears in the grid. The XMap editor automatically completes the mapping statement fields.
3. To create conditional logic in the grid, add a Router mapping statement as follows:
   a. Under the Router mapping statement, create Option mapping statements. Drag and drop input and output schema nodes into the Option statement fields in the grid.
   b. Under the Router mapping statement create one Default mapping statement to specify what happens if no Option mapping statement applies.
   c. Under the Option mapping statements, create Map mapping statements to specify conditions to map the input node to the output node.
4. To provide a common context for a group of statements, add a Group mapping statement. Nest Map mapping statements under the Group mapping statement.
5. To call another XMap object, add a Run XMap statement.
6. To change the context and logic for a mapping statement, edit mapping statement properties as follows:
   a. Demote statements to be child statements, or promote statements to be parent statements.
   b. Create XPath expressions to change the context or add predicates using the XPath editor.

Step 3. Configure the Streamer

If the transformation has XML output, JSON output, or another structured format selected as output, the wizard creates a Streamer component and XMap object in the wizard. Edit the Streamer to identify the XMap in the transformation.

1. In the **Objects** view, double-click the Streamer, a Script object, to open it in the IntelliScript editor.

2. To configure the Streamer to identify the XMap in the transformation, perform the following steps:
   a. To configure the necessary element, expand the **contains** element, in the IntelliScript editor. Couble-click the double-right arrows >> near the element.
   b. Expand the **repeating_segment** element. Double-click the double-right arrows >> near the element.
   c. In the **run_component** element select the relevant XMap object from the list of components.
3. To configure the Streamer to read data from locations in the source document and write the data to XML, right-click
4. To configure the Streamer further, within the **Streamer** element, nest **ComplexSegment** and **SimpleSegment** components corresponding to the source structure.
5. For each **SimpleSegment**, define the opening marker and closing marker if required. Define the transformation that processes the segment.

Step 4. Configure the Complex File Reader

Add a Data Processor transformation that transforms Avro to a mapping with a complex file reader. Configure the complex file reader to process Avro input.

1. In the Mapping editor, create a complex file reader object.
2. To configure the complex file reader, perform the following steps:
   a. In the **Advanced** tab of the **Properties** view, select the **File Format** property, and then choose **Custom Input**.
   b. Select the **Input Format** property, and then type `com.informatica.avro.AvroToXML`.
   c. To optimize performance, use the **Input Format Parameters** property to tune the `MaxOutputAccumulation` parameter. By default, the `MaxOutputAccumulation` parameter, which defines the expected number of output records, is set to 50,000. To change the setting to 250,000 for example, enter `"MaxOutputAccumulation"="250000"`.
   d. By default, the complex file reader adds the schema to the complex file reader output within a single element directly after the root element. If you do not want to add the schema to the output, select the **Input Format Parameters** property, then type `"InjectSchema"="false"`.

      Use a semi-colon to separate multiple parameters, for example `"MaxOutputAccumulation"="250000";"InjectSchema"="false"`.
3. Add the Data Processor transformation to the mapping. The transformation input port should remain set to binary input, the default setting for Avro input.
4. Link the complex file reader output port to the Data Processor transformation input port. The complex file reader output port should remain set to binary output.

Configure a Transformation with Avro Output

To create a Data Processor transformation for Avro output, use the Data Processor transformation wizard. The wizard creates the transformation in the Model repository with the components you need to transform Avro output. Use the XMap editor to edit the XMap, if included. Add the transformation to a mapping with a complex file writer.

1. Create the Data Processor transformation with the New Transformation wizard. Add an Avro schema or example file that defines the expect output structure.
2. If the transformation has XML, JSON, or another structured format as input, use the XMap editor to edit and customize the XMap that the wizard created in the transformation.

3. To configure the output settings for the Data Processor transformation, in the **Output Control** panel of the **Settings** view, select the relevant output option in the **Binary output port collection mode** area.

   Note: To configure the output to send one record at a time with the schema, select the **Output row for each row** option. To collect all the records with the schema into one output stream, select the **Collect input rows to a single output** option.
4. Add the Data Processor transformation to a mapping. The transformation remains set to binary output, the default setting for Avro output.
5. Create and link a complex file writer to the Data Processor transformation in the mapping to receive the Avro output from the transformation.

   Note: The input setting remains set to binary, the default setting. The complex file writer does not support compression for Avro output.
6. Configure the Complex File Writer. In the Data Object Operation, in the **Advanced** tab, for the **File Format**, select **Custom Output**. For **Output Format**, enter `com.informatica.avro.XMLToAvro`.

COBOL Processing Library

The COBOL library transforms COBOL data to and from XML. When you use the wizard to create a transformation with COBOL input or output, you select a COBOL copybook to define the expected structure of the input or output data.

When you create a Data Processor transformation with COBOL input or output with the wizard, Developer adds the following objects to the transformation:

- A schema object that defines an XML representation of the COBOL data structure.
- For COBOL input, Developer adds a Parser that transforms input data from the COBOL data definition to XML.
- For COBOL output, Developer adds a Serializer that transforms XML to COBOL.

Note: You can create a Data Processor transformation that uses COBOL input or output, but not both. To process EBCDIC encoded COBOL, ensure that you change the encoding settings for the Data Processor transformation to EBCDIC.

Creating a Transformation for COBOL

Use the Data Processor transformation wizard to create a Data Processor transformation with COBOL input or output.

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. Select **Create a data processor using a wizard** and click **Next**.
5. Select an input format and click **Next**.
6. If you select COBOL as an input format, browse to select a COBOL copybook. Click **Next**.

   The copybook specification file generally has an `*.txt` extension. Developer adds an XSD schema file representing the copybook to the Model repository.
7. Select an output format and click **Next**.

8. If you select COBOL as an output format, browse to select an COBOL copybook. Click **Next**.

   If you selected COBOL as the input format, you do not have the option to select COBOL as the output format.

9. Click **Finish**.

   The Developer tool creates the transformation in the repository. The **Overview** view appears in the Developer tool.

10. In the **Objects** view, double-click the Parser to open it in the IntelliScript editor.

11. If the COBOL data is encoded in EBCDIC, in the **Settings** view change the input or output encoding to the relevant EBCDIC codepage.

COBOL Data Definitions

The COBOL copybook that you use to create a Data Processor transformation can contain data definitions of any complexity. The COBOL copybook and input must comply with data definition rules described in this section.

Supported Data Definitions

The COBOL import supports data definitions of any complexity. For example, the data definitions can use the packed decimal (`COMP-3`), binary (`COMP-1`, `COMP-2`, or `COMP-4`), and logical decimal point (`99v99`) data types. They can contain features such as `REDEFINES`, `OCCURS`, and `OCCURS DEPENDING ON` clauses.

Data Definition Rules

A COBOL data definition must comply with the following rules:

- No more than 72 characters for each line, and no text beyond column 72
- The first line must be a remark, with a `*` in column 7, or it must start with a level number
- The first level number must be in column 1 or 8.

Unsupported Data Definitions

The Data Processor transformation does not support the following COBOL data definitions:

- The special level numbers 66, 77, and 88
- `USAGE` clauses at a group level
- `INDEXED BY` clauses
- `POINTER` and `PROCEDURE-POINTER`

Test Procedures

When you test the COBOL Parser you transform sample COBOL data to XML and verify the output. After you test the Parser, you can run the COBOL serializer on the output of the Parser.

Testing a COBOL Parser

To test the COBOL Parser you need an input file that contains sample COBOL data. The data structure must conform to the data definition that you imported. The Parser transforms sample COBOL data to XML and you verify the output.

1. In the **Object** view, double-click the COBOL Parser.

   The Parser appears in the script panel of the IntelliScript editor.

2. Right-click the Parser name, and then click **Set as Startup Component**.
3. Expand the IntelliScript tree, and then edit the `example_source` property of the Parser. Change its value from `Text` to `LocalFile`.

   The wizard configures the COBOL Parser in a way that does not require an example source document. When you finish testing, you can remove the example source. The example source has no effect on the transformation at run-time.
4. To assign an example file, expand the `LocalFile` component by clicking the double right arrows **>>**. Double-click the `file_name` property and browse to the input file that contains the sample COBOL data.
5. In the **Data Viewer** view **Input** panel, you can examine the example file. If the document does not display, right-click the Parser name in the IntelliScript, and click **Open Example Source**.
6. Click **Run** > **Run Data Viewer** to test the Parser.
7. In the **Data Viewer** view **Output** panel, examine the Parser output.
8. To confirm that the Parser ran without error, in the **Data Viewer** view **Output** panel, click the **Show Events** button. Examine the execution log in the **Data Processor Events** view.

Testing a COBOL Serializer

After you test a COBOL Parser, you can run the COBOL Serializer on the output of the Parser.

1. In the Data Transformation Explorer, double-click the TGP script file of the Serializer.

   The Serializer appears in the script panel of the IntelliScript editor.
2. Right-click the Serializer name, and then click **Set as Startup Component**.
3. Click **Run** > **Run** to activate the Serializer. At the prompt, browse to the `Results\output.xml` file, which you generated when you ran the Parser.
4. Examine the execution log in the **Events** view. Confirm that the Serializer ran without error.
5. To view the Serializer output, double-click the `Results\output.xml` file in the Data Transformation Explorer.

   The display should be the same as the original input on which you ran the Parser.

Editing a Transformation for COBOL

You can edit a transformation for COBOL that you generate with the Data Processor transformation wizard.

If you do this, document your editing. The documentation might be essential if you later revise the COBOL data definition, re-import it to a new transformation, and need to reproduce your editing.

Optimizing Large COBOL File Processing in the Hadoop Environment

You can optimize how a mapping with a complex file reader and a Data Processor transformation processes large COBOL files in the Hadoop environment.

In order to optimize large COBOL file processing, you must be able to use a regular expression to split the records. If the COBOL file can be split with a regular expression, you can define an input parameter for the complex file reader that provides a regular expression that determines how to split record processing in the Hadoop environment.

Configuring the Complex File Reader for COBOL

Add a Data Processor transformation that transforms COBOL input to a mapping with a complex file reader. Configure the complex file reader to optimize how the mapping processes COBOL input in a Hive environment. The input encoding for the complex file reader must be EBCDIC.

1. In the Mapping editor, create a complex file reader object.
2. To configure the complex file reader, perform the following steps:
   a. On the **Advanced** tab of the **Properties** view, select the **File Format** property, then choose **Input Format**.
   b. Select the **Input Format** property, then type `com.informatica.hadoop.reader.RegexInputFormat`.
   c. To optimize performance, use the **Input Format Parameters** property to define the regular expression in the form `Regex="<regular expression>"`.
3. Create a Data Processor transformation for COBOL input using the wizard.
4. Add the Data Processor transformation to the mapping. The transformation input port should remain set to binary input, the default setting for COBOL input.
5. Link the complex file reader output port to the Data Processor transformation input port. The complex file reader output port should remain set to binary output.

JSON

When you create a Data Processor transformation with JSON input or output, you select a JSON schema or sample file for the transformation. The schema or sample file defines the expected structure of the input or output data hierarchy.

JavaScript Object Notation (JSON) is a hierarchical data-interchange format similar to XML. The JSON format is often used to transmit structured data over a network connection. A Mapper or Serializer uses a JSON input schema or input document in the same way as an XML input schema and input document to define the expected input data hierarchy. A Parser uses a JSON output schema or output document to define the expected output data hierarchy.

When you use the Data Processor transformation wizard to create a transformation with JSON input or output, the transformation can contain a Parser, Mapper, Transformer, or a Serializer associated with the JSON hierarchy. The JSON schema is converted into an .xsd file that defines the hierarchical structure of the JSON file. You can use this .xsd schema with any JSON document that has the same hierarchical format.

JSON Schemas

When you create a Data Processor transformation with the wizard, you use a JSON schema or example source to define the JSON input or output hierarchy.

The Data Processor transformation wizard generates an XML schema in the Model repository that specifies the JSON structure that the transformation components use. The transformation contains a transformer associated with the schema, and can contain other components, depending on the input or output you selected in the wizard.

Scripts use schemas to define the input and output hierarchical structures. A JSON input schema must comply with the JSON Schema Internet Draft, published by the Internet Engineering Task Force.

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

- Introduction to JSON: http://www.json.org
- Convert a JSON document to a JSON Schema: http://jsonschema.net

Sample JSON Schema

The following example is a sample JSON schema:

```
{"type":"object",
    "$schema": "http://json-schema.org/draft-03/schema",
    "id": "#",
    "required":false,
    "properties":{
        "OrgId": {
            "type":"string",
            "id": "OrgId",
            "required":false
        },
        "metrics": {
            "type":"array",
            "id": "metrics",
            "required":false,
            "items": {
                "type":"object",
                "id": "0",
                "required":false,
                "properties":{
                    "name": {
                        "type":"string",
                        "id": "name",
                        "required":false
                    },
                    "valueTrend": {
                        "type":"array",
                        "id": "valueTrend",
                        "required":false,
                        "items": {
                            "type":"object",
                            "id": "0",
                            "required":false,
                            "properties":{
                                "date": {
                                    "type":"string",
                                    "id": "date",
                                    "required":false
                                },
                                "val": {
                                    "type":"string",
                                    "id": "val",
                                    "required":false
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

The schema defines the elements and attributes that can occur in a JSON document. The schema uses the JSON syntax to specify the hierarchy and sequence of elements, whether elements are required, the element type, and possible values.

The previous sample schema defines the following JSON input document:

```
{"OrgId": "ORG000000000001",
    "metrics": [
```

```
        {
            "name": "COL1",
            "valueTrend": [
                    {
                        "date": "2011-11-01",
                        "val": "122.456"
                    },
                    {
                        "date": "2011-11-02",
                        "val": "215.1"
                    }
                ]
        },
        {
            "name": "COL2",
            "valueTrend": [
                    {
                        "date": "2011-11-01",
                        "val": "122.456"
                    },
                    {
                        "date": "2011-11-02",
                        "val": "215.1"
                    }
                ]
            }
        ]
    }
```

If you trace through the schema, you can determine the relationship between the elements of the schema and input document.

The schema hierarchy contains the `metrics` object that nests the `valueTrend` array. The array contains the fields `date` and `val` that are of the `string` data type.

Creating a Transformation with JSON

Create a Data Processor transformation with JSON input or output.

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. Select **Create a data processor using a wizard** and click **Next**.
5. Select an input format and click **Next**.
6. If you select JSON as an input format, browse to select a JSON schema or sample JSON file. Click **Next**.

   Typically, the JSON file has an `*.json` extension. Developer adds an XSD schema file representing the JSON hierarchy to the Model Repository.
7. Select an output format and click **Next**.
8. If you select JSON as an output format, browse to select a JSON schema or sample JSON file. Click **Next**.

   If you selected JSON as the input format, you will not have the option to select JSON as the output format.
9. Click **Finish**.

   The Developer tool creates the transformation in the repository. The **Overview** view appears in the Developer tool.
10. In the **Objects** view, double click the transformation component to open it in the IntelliScript editor.

Parquet

Use the wizard to create a transformation with Parquet input or output. When you create a Data Processor transformation to transform the Parquet format, you select a Parquet schema or example file that defines the expected structure of the Parquet data. The wizard creates components that transform Parquet format to other formats, or from other formats to Parquet format. After the wizard creates the transformation, you can further configure the transformation to determine the mapping logic.

Apache Parquet is a columnar storage format that can be processed in a Hadoop environment. Parquet is implemented to address complex nested data structures, and uses a record shredding and assembly algorithm. For more information about Parquet, see http://parquet.incubator.apache.org/documentation/latest//.

A transformation that reads Parquet input or output relies on a schema. When the transformation reads or writes Parquet data, the transformation uses the schema to interpret the hierarchy.

After you create a Data Processor transformation for Parquet input, you add it to a mapping with a complex file reader. The complex file reader passes Parquet input to the transformation. For a Data Processor transformation with Parquet output, you add a complex file writer to the mapping to receive the output from the transformation.

Creating a Transformation with Parquet Input or Output

Create a Data Processor transformation with Parquet input or output.

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. Select **Create a data processor using a wizard** and click **Next**.
5. Select an input format and click **Next**.
6. If you select Parquet as an input format, browse to select a Parquet schema or sample Parquet file. Click **Next**.

   The Developer tool adds a schema object file representing the Parquet hierarchy to the Model repository.
7. Select an output format and click **Next**.
8. If you select Parquet as an output format, browse to select a Parquet schema or sample Parquet file. Click **Next**.

   If you selected Parquet as the input format, you will not have the option to select Parquet as the output format.
9. Click **Finish**.

   The Developer tool creates the transformation in the repository. The **Overview** view appears in the Developer tool.
10. In the **Objects** view, double click the transformation component to open it in the IntelliScript editor.

    To configure the transformation component, add mapping statements.

Configure the Complex File Reader For Parquet Input

After you create a Data Processor transformation that converts Parquet input, add the transformation to a mapping with a complex file reader. Configure the complex file reader to process Parquet input.

1. In the Mapping editor, create a complex file reader object.
2. To configure the complex file reader, perform the following steps:
   a. In the **Advanced** tab of the **Properties** view, select the **File Format** property, and then choose **Input Format**.
   b. In the **Advanced** tab, select the **Input Format** property, and then type `com.informatica.parquet.ParquetToXML`.
   c. To optimize performance, use the **Input Format Parameters** property to tune the `MaxOutputAccumulation` parameter. By default, the `MaxOutputAccumulation` parameter, which defines the expected number of output records, is set to 50,000. To change the setting to 250,000 for example, enter `"MaxOutputAccumulation"="250000"`.
   d. By default, the complex file reader adds the schema to the complex file reader output within a single element directly after the root element. If you do not want to add the schema to the output, select the **Input Format Parameters** property, then type `"InjectSchema"="false"`.

      Use a semi-colon to separate multiple parameters, for example `"MaxOutputAccumulation"="250000";"InjectSchema"="false"`.
3. Add the Data Processor transformation to the mapping. The transformation input port should remain set to binary input, the default setting for Parquet input.
4. Link the complex file reader output port to the Data Processor transformation input port. The complex file reader output port should remain set to binary output.

Configure a Transformation with Parquet Output

To create a Data Processor transformation for Parquet output, use the Data Processor transformation wizard. The wizard creates the transformation in the Model repository with the components you need to transform Parquet output. Use the XMap editor to edit the XMap, if included. Add the transformation to a mapping with a complex file writer.

1. Create the Data Processor transformation with the New Transformation wizard. Add a Parquet schema or example file that defines the expect output structure.
2. If the transformation has XML, JSON, or another structured format as input, use the XMap editor to edit and customize the XMap that the wizard created in the transformation.
3. To configure the output settings for the Data Processor transformation, in the **Output Control** panel of the **Settings** view, select the relevant output option in the **Binary output port collection mode** area.

   Note: To configure the output to send one record at a time with the schema, select the **Output row for each row** option. To collect all the records with the schema into one output stream, select the **Collect input rows to a single output** option.
4. Add the Data Processor transformation to a mapping. The transformation remains set to binary output, the default setting for Parquet output.
5. Create and link a complex file writer to the Data Processor transformation in the mapping to receive the Parquet output from the transformation.

   Note: The input setting remains set to binary, the default setting. The complex file writer does not support compression for Parquet output.
6. Configure the Complex File Writer. In the Data Object Operation, in the **Advanced** tab, for the **File Format**, select **Custom Output**. For **Output Format**, enter `com.informatica.parquet.XMLToParquet`.

XML

Use the wizard to create a transformation with XML input or output. When you create a Data Processor transformation with XML input or output, you select an .xsd schema or XML sample file for the transformation. The schema or sample file defines the expected structure of the input or output data hierarchy. If you select a sample file, the wizard creates a schema from the sample file data hierarchy.

The wizard creates a Data Processor transformation that can contain a Parser, Mapper, XMap, or a Serializer associated with the XML hierarchy. A Mapper, XMap, or Serializer uses an input schema to define the expected input data hierarchy. An XMap, Mapper, or Parser uses an output schema to define the expected output data hierarchy. For more information about the schema syntax, see http://www.w3.org.

The wizard creates the basic components or relational mapping that the transformation requires based on the input and output types. The transformation might be complete, or can serve as a starting point for further configuration.

If the transformation has structured input and output, the wizard might create an XMap that you configure to transform data from one hierarchy to another. Use the XMap editor to link the input and output schema hierarchy nodes and define the transformation logic. For more information about the XMap object and editor, see "XMap Overview" on page 88.

If the transformation has relational input or output that you want to transform from structured data or to structured data, the wizard creates a relational mapping. The **Ports** panel in the **Overview** view shows the hierarchical schema nodes and the relational ports. Use the **Ports** panel to link the hierarchical elements to the relational ports and groups. For more information about transforming relational data, see "Relational Input and Output Overview" on page 70.

Creating a Transformation that Transforms XML

Create a Data Processor transformation with XML input, XML output, or both.

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. Select **Create a data processor using a wizard** and click **Next**.
5. Select XML or another input format and click **Next**.
6. If you selected XML as the input format, browse to select a schema or sample XML file. Click **Next**.

   Developer adds an .xsd schema file representing the hierarchy to the Model Repository.
7. Select XML or another output format and click **Next**.
8. If you select XML as an output format, browse to select a schema or sample XML file. Click **Next**.
9. Click **Finish**.

   The Developer tool creates the transformation in the repository. The **Overview** view appears in the Developer tool.
10. To edit a specific component in the transformation, in the **Objects** view, double click the transformation component to open it in the IntelliScript editor.

CHAPTER 4

Relational Input and Output

This chapter includes the following topics:

- Relational Input and Output Overview, 70
- Relational Input, 70
- Relational Output, 76

Relational Input and Output Overview

A Data Processor transformation can read relational database input from input ports and transform it into other formats. A transformation can output rows of relational data to output ports. You can define the mapping with Data Processor transformation ports on the transformation Overview view. Or you can use the Data Processor transformation wizard to automatically map relational data.

You can transform relational data to hierarchical data. To transform input groups into hierarchal data, map nodes from group of relational ports to the hierarchical ports. You can pass the data from the hierarchical output ports to another transformation in the mapping.

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

Relational Input

A Data Processor transformation can convert relational input into hierarchical output. A relational database is a database that has a collection of tables of data, organized according to a relational model. Tables might have additional relationships with each other.

In the relational model, each table schema identifies a column called the primary key, to uniquely identify each row. You identify the relationship between each row in the table and a row in another table with a foreign key. A foreign key is a column in one table that points to the primary key of another table.

To convert relational port data into hierarchical data, you must define the structure of the mapping based on an hierarchical schema. You import a schema to the Model repository. After you import the schema, you can view the schema components in the Developer tool. If the hierarchical schema has multiple elements that can be a root element, choose one node to be the root element.

In the **Ports** panel of the **Overview** view, you can map the relational input ports to schema nodes. To the left side of the panel is the **Transformation input** area and to the right side of the tab is the **Service input** area.

When you drag a node from a **Service input** node to **Transformation input** port, you map from a schema node to a relational input node. The Developer tool creates the input ports to map the data. You can define groups, define ports, and map nodes from the input to the output ports.

A Data Processor transformation with relational input can contain pass-through ports. You add pass-through ports to the root group of the relational structure.

Relational Input Port Configuration

To map relational input ports to hierarchal output, select a schema to define the output. You must also define the relational input ports. On the Ports panel, create and define groups of ports, and map the nodes from the hierarchal nodes to the ports.

To define an input mapping, select the **Data processor mode** to be **Input Mapping** or **Input Mapping and Service**. The mapping uses a schema to define the output. If the schema has more than one element that can be a root element, you can choose a node from the 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 **Transformation output** area. Click **Show As Hierarchy** to display the output nodes in a hierarchy. Each child group appears underneath the parent group.

Create and define relational input ports with one of the following methods:

Drag nodes to ports

Drag nodes from the **Transformation output** area to the **Transformation input** 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 **Transformation input** 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.

Automatically create the ports

Click **Auto Map**. The Developer tool creates input groups and adds ports to the groups based on the output hierarchy.

When you drag nodes to the **Transformation input** area, the Developer tool updates the location field with the location of the node in the 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 relational group, drag an output node to an empty column in the **Transformation input** area. If you drag a multiple-occurring child node to an empty input column, the Developer tool asks you to relate the group to other groups. When you select a group, the Developer tool creates keys to relate the groups.

You can also create a new relational group by clicking **New** > **Group** in the **Transformation input** area. Enter a name for the group. Configure related groups of input ports in the **Transformation input** 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.

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

Guidelines to Link Input Ports

When you use the New Transformation wizard to auto-generate a Data Processor transformation, the Developer tool creates relational ports based on the schema hierarchy, and then links the relational ports to the hierarchical nodes.

Consider the following rules and guidelines when you link relational input ports to hierarchical output nodes:

- You can link an input relational port to a node in the hierarchy.
- Link a primary key from the relevant element or attribute in the hierarchy to a relational group in the input. The primary key identifies each row in the relational tables.
- Link a foreign key from the relevant element or attribute in the hierarchy to a relational group in the input. A foreign key in the relational input is a column in one table that points to the primary key of another table.
- The input relational port and the hierarchical node must have compatible data types.
- Keys might be of the string type or the integer type.

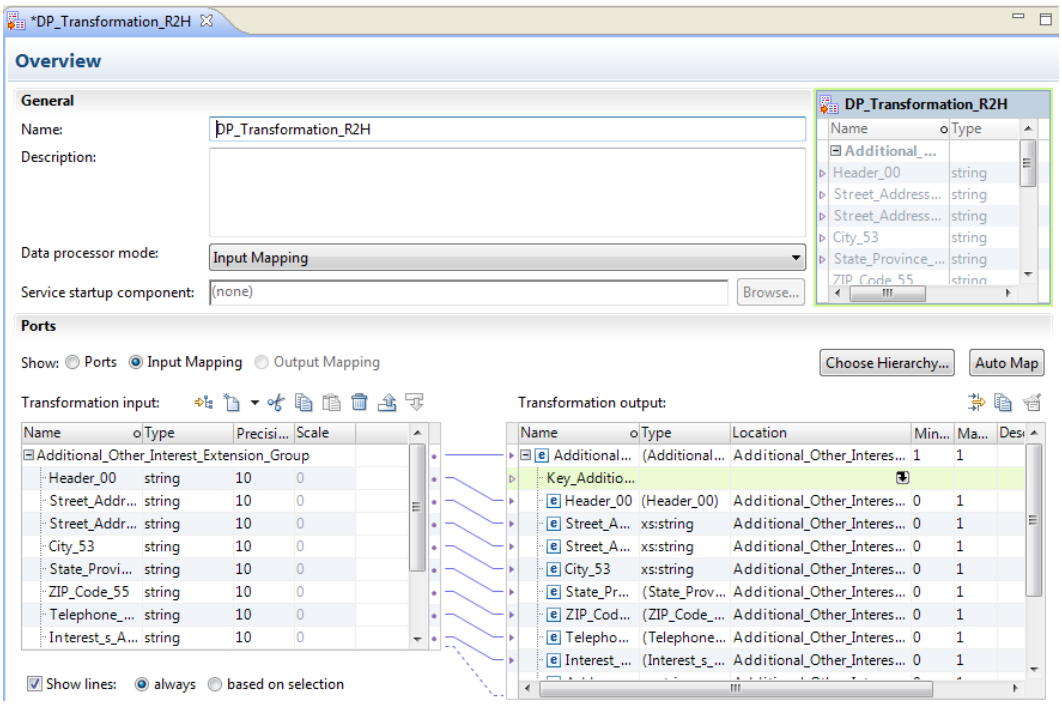
Define Input Relational Ports with the Overview View

To transform relational data to hierarchical data in the Data Processor transformation, map nodes from group of relational ports to the hierarchical nodes. Use the Overview View to link relational ports to hierarchical ports.

To transform relational input to hierarchal output, enable relational input from the **Overview** view. The Developer tool removes the default input port on the view.

Select **Input Mapping**. The **Ports** panel appears in the **Overview** view.

The following iamge shows the **Ports** panel:



To the left of the **Ports** panel is the **Transformation output** area that contains the hierarchical schema nodes. To the right is the **Transformation input** area that contains the relational elements and groups.

You can create ports in the **Transformation input** area, and link relational elements to the schema nodes. You can also drag the pointer from a node in the schema to an empty field in the **Transformation input** area to create a port. When you connect a relational port to a schema node, the Developer tool shows a link between them.

Clustering_Key Ports

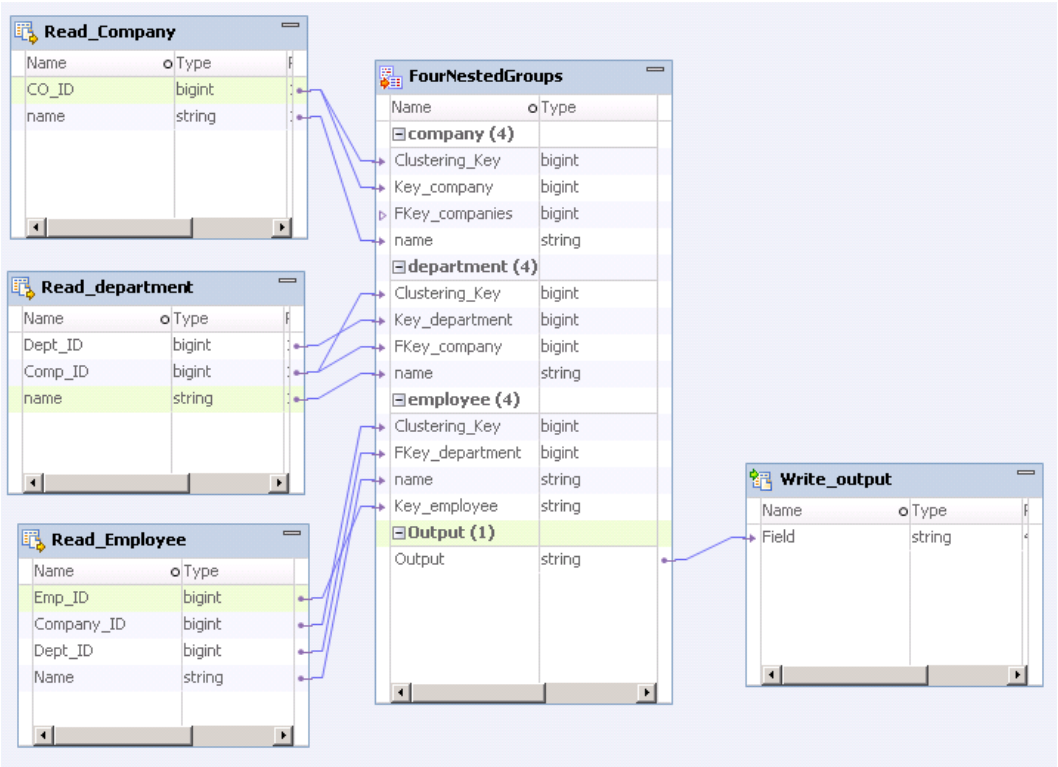
When you create a relational to hierarchical Data Processor transformation with multiple groups in the Hive environment, enable input data partitioning to ensure that data for each row processes correctly. The Data Integration System partitions the input rows according to a port that functions as a partitioning key named the Clustering_Key.

To partition input data to a Data Processor transformation in a mapping, select the transformation in the mapping, and in the **Advanced** tab of the **Properties** view, select to enable partitioning. When you enable partitioning, the Developer creates a Clustering_Key port in the Data Processor transformation for each input group.

Each input group must use the same foreign key to the input root group to help partitioning. To sort data according to a key, connect the selected foreign key relational input port of each Data object to the relevant Clustering_Key port in the Data Processor transformation. The Data Integration Service uses the Clustering_Key to partition and process the data.

You must use the same key in all of the relational input groups. If needed, you can use a Joiner transformation to add the key to a relational input group that does not have that key.

The following image shows a mapping with the foreign key Company_ID in the relational input groups linked to the Clustering_Key ports in the Data Processor transformation:



Normalized Relational Input

When you normalize the relational input data in the hierarchical output, the data values do not repeat in the hierarchical group. You create a one-to-one relationship between the hierarchy levels in the hierarchal output data and the input groups of ports.

Normalized Relational Input Example

You want to transform relational input with a group that contains details of managers from several companies to separate XML hierarchies. In the input, each manager record contains company details. In the output, one XML hierarchy contains details for companies, and a separate XML hierarchy contains details for managers.

In the relational input, the Company_ID and Company_Name elements repeat for each manager in the company:

Company_ID	Company_Name	Manager_ID	Manager_Name
100	Percy Accounting	76500	Cindy Jacques
100	Percy Accounting	46501	Tom Jorry
100	Percy Accounting	86509	Delilah Smith

If you define the XML output to contain a Company parent hierarchy level and an Managers child hierarchy level, you might use the following hierarchy groups:

```
Companies
    Company_Key
    Company_ID
    Company_Name


Managers
    Company_Key
    Manager_ID
    Manager_Name
```

The Company_Key element relates the Managers hierarchy to the Companies hierarchy.

Pivoted Relational Input

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

To pivot multiple-occurring elements, map the multiple-occurring port element to specific output nodes.

Pivoted Relational Input Example

You want to transform relational input that contains a table of telephone numbers to an XML hierarchy with separate elements for different types of telephone numbers.

In the relational input, the Telephone_type element defines the type of phone number listed for each person:

Telephone_Number	Telephone_Type	Last_Name	First_Name
9173327437	Mobile	Sandrine	Jacques
9174562342	Mobile	Race	Tom
8484526471	Home	Race	Tom
7023847265	Work	Smith	Delilah
9174596725	Mobile	Smith	Delilah

In the Telephones output XML hierarchy, different types of telephone numbers in the parent group have separate elements:

```
Telephones
    Telephone_Number
```

```
        Last_Name
                    First_Name
        Work_Telephone
        Mobile_Telephone
        Home_Telephone
```

Denormalized Relational Input

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

To denormalize input data, map nodes from a group of ports to a child hierarchy level. All the elements repeat at the child hierarchy level.

Denormalized Relational Input Example

You want to transform relational input with separate groups for manager details and company details into a JSON hierarchy that contains both manager and company details.

The Company_Name element does not appear in the group with manager details. The Company_ID element is the foreign key in the first relational group.

Company_ID	Manager_ID	Manager_Name
100	56673	Kathy Jason
100	23501	Jackie Lyons
100	44509	Bob Terrence

The second relational group contains company details.

Company_ID	Company_Name
100	Percy Accounting
102	Sandy Auto Sales
410	Movers Inc.

The Managers element in the JSON output contains both the Company_ID and the Company_Name elements:

```
Managers
    Company_ID
    Company_Name
    Manager_ID
    Manager_Name
```

The Company_ID and Company_Name elements repeat for each manager in the department.

Mapping Relational Ports to Hierarchical Nodes

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

1. To add a schema, in the **References** view , click **Add**. Browse to and select a schema.
2. To create an input mapping, in the **Overview** view **General** area, select **Input Mapping** for the Data Processor mode.
3. In the **Ports** area select **Input Mapping**.
4. Select a node as a root.
5. To define a mapping automatically, click **Auto Map**. To manually define a mapping, you define a primary key for the root input group, then define input groups and ports.
6. To add an input group or port to the **Transformation input** area, use one of the following methods:

- Drag a hierarchical group node or a child node in the **Transformation output** area to an empty column in the **Transformation input** area. If the node is a group node, the Developer tool adds a relational group without ports.
- To add a relational group, select a row and right-click to select **New** > **Group**.
- To add a relational port, right-click to select **New** > **Field**.

7. To map nodes as a primary key, use one of the following methods:
   - Select two or more hierarchy nodes and drag them to a key in the **Transformation input** area.
   - Click the **Location** column of a key in the **Transformation output** area and then select the relational input port in the **Transformation input** area.
8. To clear the hierarchical node settings for locations of ports, use one of the following methods:
   - Select one or more nodes in the **Transformation output** area, right-click and select **Clear**.
   - Select one or more lines that connect the relational input ports to the hierarchical nodes in the **Transformation output** area, right-click and select **Delete**.

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.

Relational Output Port Configuration

To transform hierarchical input to relational output, select a schema to define the hierarchical data. You must also define the relational output ports. On the Ports panel, define groups of ports and map the nodes from the hierarchical schema to the ports.

To define an output mapping, select the **Data processor mode** to be **Output Mapping** or **Output Mapping and Service**.

The mapping uses a schema to define the hierarchical input. If the schema has more than one element that can be a root element, choose a node to be the root element. To define 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 **Transformation input** area.

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

Create ports with one of the following methods:

Drag nodes to ports

Drag nodes from the **Transformation input** area to the **Transformation output** 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 **Transformation output** 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 **Transformation output** area, the Developer tool updates the location field with the location of the node in the 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 column in the **Transformation output** area. If you drag a multiple-occurring child node to an empty input or 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**. Enter a name for the group.

Configure related groups of output ports in the **Transformation output** 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.

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

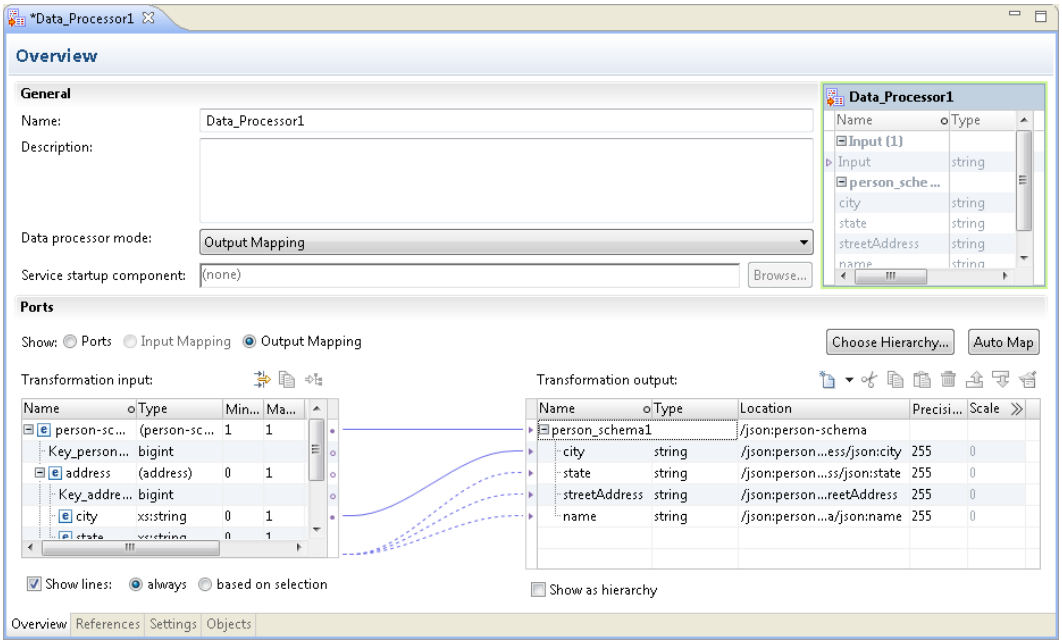
Define Output Relational Ports with the Overview View

To convert hierarchical data to relational output in the Data Processor transformation, link hierarchical nodes to relational ports. Use the Overview View to link relational ports to hierarchical ports. You create groups of output ports by linking nodes from the hierarchical output to groups of ports.

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 image shows the **Ports** panel:



The **Transformation input** area, which shows the output schema, is to the left. The **Transformation output** area, which shows the relational output ports, is to the right.

Define relational output ports in the **Transformation output** area and link 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 **Transformation output** area to create a port. When you drag a node from the output schema to a port, the Developer tool shows a link between them.

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 input hierarchal data and the output groups of ports.

Normalized Relational Output Example

You want to transform a JSON hierarchy that defines department and employee details into relational output with separate groups for department and employee details.

The JSON input contains a Staff hierarchy with elements that contain employee and department details.

```
Staff
    Department_ID
    Department_Name
    Employee_ID
    Employee_Name
```

You might create the following groups of relational ports:

Department_Key	Employee_ID	Employee_Name
100	2673	Jason Stuart
100	1501	Lila Rose
100	4309	Sarah Jacobs

Department_Key	Department_ID	Department_Name
100	1982	Accounting
102	3297	Sales
410	8276	Logistics

The Department_Key is a generated key that relates the Employees group to a Department group in the output.

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.

Pivoted Relational Output Example

You want to transform an XML hierarchy that contains employee details with multiple IDs to a relational output group with separate output elements for the employee IDs.

The following example shows two instances of Employee_ID in the Departments relational output 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, link nodes from the parent hierarchy level to the child group of output ports.

Denormalized Relational Output Example

You want to transform an XML hierarchy with separate groups for employee details and department details into a relational group that contains both employee and department details.

The XML hierarchies separate the Department details from the Employee details:

```
Department
    Department_ID
    Department_Name


Employees
    Department_ID
    Employee_ID
    Employee_Name
```

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

Department_ID	Department_Name	Employee_ID	Employee_Name
100	Accounting	56500	Kathy Jones
100	Accounting	56501	Tom Lyons
100	Accounting	56509	Bob Smith

The Department_ID and Department_Name elements repeat for each employee in the department.

CHAPTER 5

Using the IntelliScript Editor

This chapter includes the following topics:

- IntelliScript Editor Overview, 80
- Opening an IntelliScript Editor, 81
- Editing Procedures, 82
- IntelliScript Editor Menus, 86

IntelliScript Editor Overview

The IntelliScript editor is the main editor where you can configure a Script component, such as Parser, Mapper, or Serializer.

The IntelliScript editor has a tree structure. The top, global level of the tree displays the runnable transformation components such as Parsers. These components are called runnable because you can set them as the startup component of the transformation.

You can also define non-runnable components, such as variables or actions, at the global level. This allows you to use the components at other locations in the project.

At the nested levels of the IntelliScript editor, you can define components such as anchors and actions. You can click the + or – symbols to expand the IntelliScript editor tree and display the nested levels.

Creating a Script

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

1. In the Data Processor transformation **Objects** view, click **New.**
2. Enter a name for the Script 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 uses the text to test a Script.
10. Click **Finish.**

    The **Script** view appears in the Developer tool editor.

Opening an IntelliScript Editor

The IntelliScript editor displays a Script component in the Data Processor transformation. To open an IntelliScript editor, double-click a Script component in the Data Processor transformation. You can view the Data Processor transformation components in the **Outline** tab.

IntelliScript and Data Viewer

The IntelliScript editor displays one or more Script components. This is where you define the transformation components.

You can view the example source document of a transformation in the **Data Viewer** tab. You can use this pane to help view, configure, or test a transformation

The **Data Viewer** example pane is read-only. You cannot edit the example source document in Data Transformation Studio.

You can view the Script in script mode, with a code representation of the Script, or in IntelliMode, using the IntelliScript Editor. By default, you view the Script in IntelliMode.

Finding Anchors

When you edit a parser, it is easy to find the corresponding anchors in the IntelliScript and example panes.

- Right-click an anchor in the IntelliScript, and click **View Marking**. This finds the anchor in the example pane.
- Click an anchor in the example pane. The anchor is automatically selected in the IntelliScript.

Components and Properties

The IntelliScript contains two kinds of items:

- Components. Items that you can insert and delete in the IntelliScript tree. Some examples of components are parsers, serializers, anchors, actions, and transformers.
- Properties. Items that you can edit, but you cannot insert or delete, except by inserting or deleting a component that contains them. Some examples of properties are the `example_source` property of a parser or the `search` property of a marker anchor.

The IntelliScript editor displays the components and properties in different colors.

Basic and Advanced Properties

To help simplify the display, the IntelliScript organizes the properties in two categories:

- Basic properties. Properties that are important in most uses of the component. You usually need to assign these properties to use the component.
- Advanced properties. Properties that you often do not need to assign. The properties may have default values that you usually do not need to change, or the properties may implement options that you often do not need to use.

The IntelliScript always displays the basic properties. It hides the advanced properties until you choose to display them.

The distinction is for display purposes only. Advanced properties are just as easy to use as basic properties. Do not hesitate to use them if they are needed in your projects.

Displaying the Advanced Properties of a Component

▶ Click the **>>** icon at the right of the component name.

The **>>** icon changes to **<<** and the advanced properties appear.

Hiding the Advanced Properties

▶ Click the **<<** icon at the right of the component name.

The **<<** icon changes back to **>>** and the advanced properties disappear. However, if you assigned a non-default value to an advanced property, it remains visible.

Editing Procedures

To edit the IntelliScript, follow the procedures described in this section.

Basic Procedure for Editing

To edit the IntelliScript:

1. Click the component or property that you want to edit.
2. Press **ENTER** to enter the editing mode. In most locations, you can also double-click instead of pressing **ENTER**.
3. Assign the component name or the property value.
4. Press **ENTER** again to complete the editing operation.

Copy and Paste

You can copy and paste components in the IntelliScript.

To copy multiple components simultaneously, press **CTRL** or **SHIFT** while you select with the mouse. The components must all be at the same level of nesting in the IntelliScript.

You can paste components only in locations where they make sense. For example, you can paste serialization anchors under a serializer but not under a parser.

Drag and Drop

You can move components from one location to another by dragging with the mouse. For example, you can use this method to alter the sequence of anchors within a parser.

To move multiple components, press **CTRL** or **SHIFT** while you select with the mouse. Release **CTRL** or **SHIFT** and then drag with the mouse.

To copy the selected components, instead of moving them, hold **SHIFT** down while you drag.

Find and Replace

To find or replace text in the IntelliScript, select **Edit** > **Find** or **Edit** > **Replace**.

Inserting Components in the IntelliScript

Under many components, the IntelliScript displays a horizontal line, usually bearing a label such as `contains`. The line is followed by an arrow and three dots (. . .). You can insert nested components at the three dots.

Inserting a Component

1. Select the three dots and press **ENTER**.

   A list of the components appears

2. Select a component from the list.

   Alternatively, start typing the component name. The name auto-completes after you type the first few letters.

3. Press **ENTER** again to complete the insertion.

Deleting a Component

▶ Select the component and press **DELETE**.

Editing the Properties of a Component

1. Select the value of a property and press **ENTER**.

   Depending on the type of property, a text box, list, or dialog box appears.

2. Type or select the new property value.
3. Press **ENTER** to complete the assignment.

Inserting Tabs, Newlines, and Other Special Characters

When you assign a textual property, you can insert special characters by typing their numeric ASCII codes.

1. Select a property and press **ENTER** to start editing.
2. Press **CTRL+A**.

   A small dot appears, indicating that the character is an ASCII code.

3. Type the three-digit ASCII code. For example, you can type:

ASCII Code	Character
009	Tab
010	New line
013	Carriage return

4. To enter a string of ASCII codes, repeat steps 2 and 3.

   You can intersperse ASCII codes and regular text.

5. Press **ENTER** to complete the editing.

   A tab appears as a « symbol. Other characters appear as their ASCII character codes.

Defining a Global Component

You can insert components in either a global or local scope.

Scope	Description
Global scope	The component is defined at the top level of the IntelliScript. It can be accessed or used at any location in the project.
Local scope	The component is defined at a nested level of the IntelliScript. It can be accessed or used only at the particular nested location.

Most Data Transformation components can be either global or local.

For example, anchors are usually defined locally. You might define an anchor as a global component, however, if you want to use the same anchor configuration in several parsers or several times in the same parser. In each desired location, you can reference the globally-defined anchor by its identifier.

A parser can then use `MyMarker`, instead of repeating the configuration of the marker anchor every time it is needed. You can select `MyMarker` from the component list at the appropriate location within the parser, or you can drag `MyMarker` to the location.

Naming Restrictions

The names that you assign to IntelliScript components must contain only English characters (`A-Z`, `a-z`), numerals (`0-9`), and underscores (_). They must begin with a letter. They can be up to 127 characters long.

Viewing Help About a Component

You can view the online help topics describing a component or property while you edit the IntelliScript.

1. Display the **Help** view.
2. Select the component or property.

   The help scrolls to the location that describes the selected item.

IntelliScript Icons

The IntelliScript displays each component type with a characteristic icon. The following table describes the most common icons that appear in the IntelliScript.

Icon	Component
	`Parser`
	`Serializer`
	`Mapper`
	`Transformer`
	`Marker`
	`Content` `ContentSerializer`
	`Group`
	`RepeatingGroup`
	`StringSerializer`
	`Handle`
	`Key`
	Other anchors
	Actions
	Default icon, used when there is no specific icon for a component

Saving the IntelliScript

If an asterisk (*) appears on the title tab of an IntelliScript editor, the editor has unsaved changes. If you attempt to close the editor or exit with unsaved changes, you are prompted to save the Script.

IntelliScript Editor Menus

Right-click in an IntelliScript editor to display a menu. The menu options depend upon the context in which you click.

Table 1. Menu of the IntelliScript Pane

Option	Description
View Marking	Highlights the selected anchor in the example source.
Set as Setup Component	Defines the selected component as the startup component of the project.
Cut	Allows you to cut components in the IntelliScript.
Copy	Allows you to copy components in the IntelliScript.
Paste	Allows you to paste components in the IntelliScript.
Insert	Allows you to insert components in the IntelliScript.
Delete	Allows you to delete components in the IntelliScript.
Make Optional	Selects the `optional` property of a component. If a component is optional, a failure of the component does not cause its parent component to fail. For more information, see the *Data Transformation Studio User Guide*.
Make Mandatory	Deselects the `optional` property of a component. If a component is optional, a failure of the component does not cause its parent component to fail. For more information, see the *Data Transformation Studio User Guide*.
Enable	Selects the `disabled` property of a component. A disabled component is ignored. This feature is useful to disable a component temporarily for testing and debugging.
Disable	Deselects the `disabled` property of a component. A disabled component is ignored. This feature is useful to disable a component temporarily for testing and debugging.
Script Mode	The Script mode displays the raw content of the `*.tgp` file. This mode is intended for advanced troubleshooting only.
Intelli Mode	The Intelli mode displays the IntelliScript in a readable, graphical representation. This is the mode illustrated throughout this book and the other Data Transformation documentation.

Option	Description
Open Example Source	Opens the example source file of the selected parser, serializer, or mapper.
Create Serializer	Creates a serializer from the selected parser. The serializer and the parser perform inverse transformations.

Table 2. Menu of the Example Pane

Option	Description
Copy	Copies a string to the clipboard.
Insert Marker	Defines the selected text as a `Marker` anchor. The anchor is added to the IntelliScript.
Insert Content	Defines the selected text as a `Content` anchor. The anchor is added to the IntelliScript.
Insert Offset Content	Defines the selected location as a `Content` anchor. The anchor is added to the IntelliScript.
Insert RepeatingGroup	Defines the selected text as the separator of a `RepeatingGroup` anchor. The anchor is added to the IntelliScript.
View Instance	Finds the selected anchor in the IntelliScript.
View Event	Finds the corresponding event in the **Events** view.
Find	Finds a string in the example source document.
Logical Encoding	If the example source contains text in a right-to-left language, such as Hebrew or Arabic, this command toggles the display from left-to-right to right-to-left.
Line Wrap	Wraps long lines.
Save Source As	Saves the example source in a specified location under a specified name.

CHAPTER 6

XMap

This chapter includes the following topics:

- XMap Overview, 88
- XMap Schemas, 89
- Mapping Statements, 90
- XPath Expressions, 106
- XMap Variables, 113
- XMap Example, 113

XMap Overview

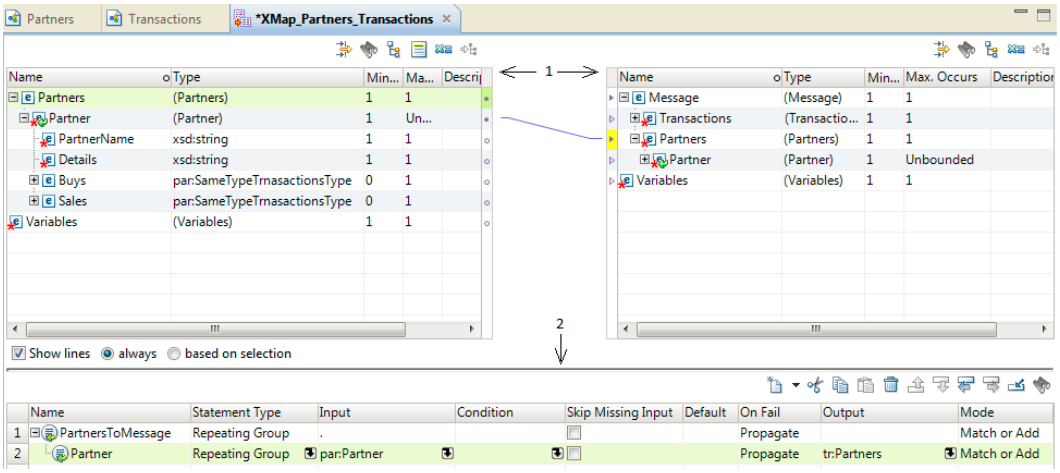
An XMap is a Data Processor transformation object that changes a hierarchical input document into another hierarchical document with a different hierarchy structure.

An XMap uses input and output schemas to define the expected hierarchy of input and output documents. Use the XMap editor to define and manage mapping statements. The XMap editor contains the input schema hierarchy and the output schema hierarchy. Mapping statements link input schema elements to output schema elements.

An XMap can transform any input hierarchical document whose elements match the input schema hierarchy into an output document with the hierarchy of the output schema.

For example, an XMap can transform customer invoices to a list of customer orders filtered by month of the year. The input is an JSON document that contains a hierarchy of customer elements. The output is an XML document that contains a hierarchy of month elements. The output XML document groups customer order by month and includes contact information and order totals.

The following image shows the XMap editor.



1. The XMap editor contains input and output hierarchical schemas. Drag and drop between the schema elements to create mapping statements.
2. The XMap editor grid shows mapping statements. Use the grid to manage and edit the mapping statements.

An XMap uses mapping statements to define how to transform an input schema element to an output schema element. You can drag from a node in the input schema to a node in the output schema to create a link. When you create links, these are mapping statements. The XMap editor shows the mapping statement in the grid.

You can edit the mapping statements in the grid. You can define conditions to transform and filter the data according to different mapping statement types and XPath expressions. XPath is a query language used to select nodes in a hierarchical document and perform computations.

You can use XPath expressions to define the context for a mapping statement. You can also add various arithmetic calculations to a mapping statement using XPath expressions.

XMap Schemas

An XMap requires hierarchical schemas that define the input and output hierarchical hierarchies. An XMap uses input and output hierarchical schemas to determine what type of data is expected in the input source document and output document. You link input and output schema nodes to create mapping statements.

A schema element is the basic building block of a mapping statement. When you define a mapping statement you might want to use a series of input schema elements in the statement, or add a variable. You can change the input and output root, add variables, and customize the schema view, but cannot edit the schema.

You can use the following options to manage the schemas and search for elements:

Customize view

Changes the way the schema appears so you can quickly search for relevant elements. You can search for a sequence of nodes, all nodes, or a choice of nodes. View these nodes to understand the schema logic. This view 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.

Choose the example source

Define the example source file. Use an example source to test the transformation and to test XPath expressions.

Mapping Statements

A mapping statement determines how to map data from the input hierarchical document to the output hierarchical document. When you drag a node from the input schema to the output schema, the XMap editor creates mapping statements in the grid.

You can use the grid to create simple or detailed mapping statements. You can drag elements from the input or output schemas into fields in the grid to include them in mapping statements.

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

Add XPath expressions to determine the context or add computations to a mapping statement. When an XPath expression identifies the context for a mapping statement, the Data Processor transformation runs the mapping statement for each occurrence of the input element or expression in the input document.

Nest mapping statements to make them dependent on other mapping statements. A mapping statement can be a parent to a group of child statements. Each time that the Data Processor transformation runs the parent statement, it runs the child statements also. Child statements appear indented from the parent in the XMap editor.

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.

RunMapplet

Calls a mapplet from the Data Processor transformation.

MappletInput

One or more MappletInput statements can be nested under the RunMapplet statement. Values are mapped to the mapplet input ports in the same order that they are listed in the MappletInput statements.

MappletOutput

One or more MappletOutput statements can be nested under the RunMapplet statement. The values in the mapplet output ports are mapped to the MappletOutput statement in the same order that they are listed in the mapplet ports.

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.

Map Statements

A Map statement is the basic building block of an XMap object and maps a simple input value to a simple output value. The input must be a single value or a constant value. You must define the input and output in a Map statement.

When you drag and drop between a simple, non-repeating input schema node and a simple, non-repeating output schema node, a Map statement is automatically created.

A Group, Repeating Group, Option or Default statement can contain one or more Map child statements.

Map Statement Properties

A Map statement contains properties that you can configure to customize the statement. You can configure the input, output, and a condition for mapping an input element to an output element.

A Map statement has the following properties:

Condition

Optional. 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.

Default

Optional. 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.

Input

Required. An XPath expression that defines the input element. The expression can evaluate to a node or value.

Mode

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

- Add. Creates an element in the output hierarchical document. If the element in not multiple-occurring, and the same values exists in the output, the mapping statement fails.
- Match. The statement expects to find a match for the element in the output elements. The statement fails if the element does not exist in the output hierarchical document.
- Match or Add. If a matching element exists in the output hierarchical document, the Data Processor transformation does not add an output element. If the element does not exist in the output hierarchical document, the transformation creates an output element.

Name

Optional. 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.

On Fail

Required. Determines the action taken if the statement fails. Choose one of the following options:

- Skip. If the statement fails, skip the statement.
- Propagate. If the statement fails, force the parent statement to also fail.

Output

Required. An XPath expression that defines the value of the element in the output hierarchical based on the results of the Input XPath expression.

Skip Missing Input

Optional. Determines whether to skip the statement if there is no match for the Input value. Choose one of the following options:

- Enabled. If the element is not in the input hierarchical document, the Data Processor transformation skips the statement without error.
- Disabled. The statement fails when the element is not in the input hierarchical document.

Statement Type

Required. Identifies the statement as a Map statement.

Group Statements

A Group statement contains a logical group of statements. A parent Group statement contains child statements. The child statements are nested underneath the Group statement in the XMap editor grid.

You can use a Group statement to provide a common context or common condition for success or failure to a group of statements. You can use a Group mapping statement if you want a set of statements to either all pass or all fail. You can use a Group mapping statement to group a set of statements to organize and simplify an XMap grid.

When you link between a complex single-occurring element to a complex single- or multiple-occurring element, the XMap editor creates a Group statement. A single-occurring element has a **Max. Occurs** value of **1**.

Group Statement Example

You want to map from an input hierarchical document with manager employee data to an output hierarchical document with worker data. There is only one manager so the input employee element is single-occurring.

A Group mapping statement is used when one element is a complex, single-occurring element. The schema has a single occurring Employee element. Employee has FirstName and LastName child elements:

```
Employee
   FirstName
   LastName
```

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:

	Name	Statement...	Input
1	Employee to Worker	Group	Employee
2	FirstName to FirstName	Map	ns0:FirstName
3	LastName to LastName	Map	ns0:LastName

The Input column for the Group statement shows that the input is the parent element 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.

Group Statement Properties

The Group statement contains properties that you can configure to customize the statement. You can configure the input, output, and a condition for mapping an input element to an output element.

The Group statement has the following properties:

Condition

Optional. An XPath expression that defines the entry condition for the Group statement and all its child statements. 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.

Input

Optional. An XPath expression that evaluates to zero or one element or value. If left empty, the statement uses the current context. If the expression evaluates to more than one value, the first is used.

Mode

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

- Add. Creates an element in the output hierarchical document. If the element in not multiple-occurring, and the same values exists in the output, the mapping statement fails.
- Match. The statement expects to find a match for the element in the output elements. The statement fails if the element does not exist in the output hierarchical document.
- Match or Add. If a matching element exists in the output hierarchical document, the Data Processor transformation does not add an output element. If the element does not exist in the output hierarchical document, the transformation creates an output element.

Name

Optional. 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.

On Fail

Optional. Determines the action taken if the statement fails. Choose one of the following options:

- Skip. If the statement fails, skip the statement.
- Propagate. If the statement fails, force the parent statement to also fail.

Output

Optional. An XPath expression that defines the value of the element in the output hierarchical based on the results of the Input XPath expression. If left empty, the statement uses the current context.

Skip Missing Input

Optional. Determines whether to skip the statement if there is no match for the Input value. Choose one of the following options:

- Enabled. If the element is not in the input hierarchical document, the Data Processor transformation skips the statement without error.
- Disabled. The statement fails when the element is not in the input hierarchical document.

Statement Type

Required. Identifies the statement as a Group statement.

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 elements or values.

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

When you link between a repeating input schema element and a repeating output schema element, the XMap editor creates a Repeating Group statement in the grid. A element repeats if the **Max. Occurs** value for the element is greater than **1**.

Repeating Group Statement Example

An input schema has the following hierarchy:

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

When you drag Employee to an output element 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 hierarchical document.

Repeating Group Statement Properties

The Repeating Group statement contains properties that you can configure to customize the statement. You can configure the input, output, and a condition for mapping an input element to an output element.

The Repeating Group statement has the following properties:

Condition

Optional. 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.

Input

Required. An XPath expression that evaluates to a sequence of nodes or values.

Mode

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

- Add. Creates an element in the output hierarchical document. If the element in not multiple-occurring, and the same values exists in the output, the mapping statement fails.
- Match. The statement expects to find a match for the element in the output elements. The statement fails if the element does not exist in the output hierarchical document.
- Match or Add. If a matching element exists in the output hierarchical document, the Data Processor transformation does not add an output element. If the element does not exist in the output hierarchical document, the transformation creates an output element.

Name

Optional. 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.

On Fail

Optional. Determines the action taken if the statement fails. Choose one of the following options:

- Skip Iteration. If a statement nested within the Repeating Group fails, and its On Fail property is set to **Propagate**, the current iteration of the Repeating Group is skipped.

- Propagate. If the statement fails, force the parent statement to also fail.

Output

Optional. An XPath expression that defines the value of the node in the output hierarchical based on the results of the Input XPath expression.

Skip Missing Input

Optional. Determines whether to skip the statement if there is no match for the Input value. Choose one of the following options:

- Enabled. If the element is not in the input hierarchical document, the Data Processor transformation skips the statement without error.
- Disabled. The statement fails when the element is not in the input hierarchical document.

Statement Type

Required. Identifies the statement as a Repeating Group statement.

Router Statements

A Router statement provides alternatives for the mapping logic based on conditions in the input document.

The Router statement contains one or more Option statements and can contain one Default statement. When the Data Processor transformation performs a Router statement, it tests each Option nested below the Router statement.

The first Option statement that matches is performed. The Option statement can contain one or more child statements of any type. If no Option statement matches, the Default statement is performed. If there is no Default statement, the Router statement fails.

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

When the transformation does not accept any Option statement and there is no Default statement, the Router statement fails. If the Option statement has a condition that is true but the mapping statements inside it fail and propagates the failure, the Router fails. You can configure the mapping to skip the Router if the Router fails.

If a Router contains no Option statement but does contain a Default statement, then the Default statement is always performed.

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 statement has one Option statement. The Option statement contains a condition that checks if the value of Role is equal to the value of Manager. If the role equals manager, the Option statement evaluates to true. The mapping evaluates the Run XMap statement nested under the Option statement. 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:

Employee to Worker	Repeating Group	ns0:Employee			Skip Iterati...	
Employee	Router				Skip	
Employee to Manager	Option		ns0:Role = "Manager"	☑	Propagate	
EmployeeToWorker	EmployeeToWorker	.			Propagate	ns0:Manager
Employee to Worker	Default				Propagate	
EmployeeToWorker	EmployeeToWorker	.			Propagate	ns0:Worker

Router Statement Properties

The Router statement contains properties that you can configure to customize the statement. You can configure the input, output, and a condition for mapping an input element to an output element.

The Router statement has the following properties:

Condition

Optional. 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.

Default

Required. 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.

Input

Required. An XPath expression that evaluates to a sequence of nodes or values.

Mode

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

- Add. Creates an element in the output hierarchical document. If the element in not multiple-occurring, and the same values exists in the output, the mapping statement fails.
- Match. The statement expects to find a match for the element in the output elements. The statement fails if the element does not exist in the output hierarchical document.
- Match or Add. If a matching element exists in the output hierarchical document, the Data Processor transformation does not add an output element. If the element does not exist in the output hierarchical document, the transformation creates an output element.

Name

Optional. 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.

On Fail

Optional. Determines the action taken if the statement fails. Choose one of the following options:

- Skip. If the statement fails, skip the statement.
- Propagate. If the statement fails, force the parent statement to also fail.

Output

Required. An XPath expression that defines the value of the element in the output hierarchical based on the results of the Input XPath expression. The Output field provides the context for the child statements.

Statement Type

Required. Identifies the statement as a Router statement.

Option Statements

The Option statement provides the condition to map the input node to the output node. An Option statement must be nested below a Router statement. The Router statement must have an Input XPath expression or a Condition XPath expression. Or, the Option statement can include an Input XPath expression and a Condition XPath expression.

The Data Processor transformation accepts an Option statement when the results of the Input field expression and a Condition field expression evaluate to a single node. If you do not define an Input field expression, the Data Processor transformation accepts the Option statement when the Condition field expression evaluates to true.

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.

Option Statement Properties

The Option statement contains properties that you can configure to customize the statement. You can configure the input, output, and a condition for mapping an input element to an output element.

The Option statement has the following properties:

Condition

Required if Input is not defined. An XPath expression that defines a condition for mapping the element. A condition is similar to a predicate expression in the Input column. The Data Processor transformation applies the Condition XPath to the result of the Input XPath.

Input

Required if Condition is not defined. An XPath expression that defines the input element. The expression can evaluate to a node or value.

Mode

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

- Add. Creates an element in the output hierarchical document. If the element in not multiple-occurring, and the same values exists in the output, the mapping statement fails.
- Match. The statement expects to find a match for the element in the output elements. The statement fails if the element does not exist in the output hierarchical document.
- Match or Add. If a matching element exists in the output hierarchical document, the Data Processor transformation does not add an output element. If the element does not exist in the output hierarchical document, the transformation creates an output element.

Name

Optional. 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.

On Fail

Optional. Determines the action taken if the statement fails. Choose one of the following options:

- Skip. If the statement fails, skip the statement.
- Propagate. If the statement fails, force the parent statement to also fail.

Output

Optional. An XPath expression that defines the value of the element in the output hierarchical based on the results of the Input XPath expression.

Statement Type

Required. Identifies the statement as a Option statement.

Default Statements

A Default statement is a child statement of a Router statement. The Router statement contains one or more Option statements and can contain one Default statement. The Data Processor transformation performs the Default statement when none of the Option statements apply.

You can define only one Default statement in a Router statement group. The Default statement must be the last statement for the Router statement group. The Default statement cannot have Input or Condition XPath expressions.

Default Statement Properties

The Default statement contains properties that you can configure to customize the statement. You can configure the default value if an input element is missing.

The Default statement has the following properties:

Default

Required. 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.

Mode

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

- Add. Creates an element in the output hierarchical document. If the element in not multiple-occurring, and the same values exists in the output, the mapping statement fails.
- Match. The statement expects to find a match for the element in the output elements. The statement fails if the element does not exist in the output hierarchical document.
- Match or Add. If a matching element exists in the output hierarchical document, the Data Processor transformation does not add an output element. If the element does not exist in the output hierarchical document, the transformation creates an output element.

Name

Optional. 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.

Output

Optional. An XPath expression that defines the value of the node in the output hierarchical based on the results of the Input XPath expression.

Statement Type

Required. Identifies the statement as a Default statement.

Run XMap Statements

A Run XMap statement calls 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. You can call an XMap to perform mapping logic that is repeated.

Run XMap Statement Properties

The Run XMap statement contains properties that you can configure to customize the statement. You can configure the input, output, and a condition for mapping an input element to an output element.

The Run XMap statement has the following properties:

Condition

Optional. 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.

Input

Optional. 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.

Mode

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

- Add. Creates an element in the output hierarchical document. If the element in not multiple-occurring, and the same values exists in the output, the mapping statement fails.
- Match. The statement expects to find a match for the element in the output elements. The statement fails if the element does not exist in the output hierarchical document.
- Match or Add. If a matching element exists in the output hierarchical document, the Data Processor transformation does not add an output element. If the element does not exist in the output hierarchical document, the transformation creates an output element.

Name

Optional. 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.

On Fail

Required. Determines the action taken if the statement fails. Choose one of the following options:

- Skip. If the statement fails, skip the statement.
- Propagate. If the statement fails, force the parent statement to also fail.

Output

Optional. An XPath expression that defines the value of the node in the output hierarchical based on the results of the Input XPath expression.

Skip Missing Input

Required. Determines whether to skip the statement if there is no match for the Input value. Choose one of the following options:

- Enabled. If the element is not in the input hierarchical document, the Data Processor transformation skips the statement without error.
- Disabled. The statement fails when the element is not in the input hierarchical document.

Statement Type

Required. Identifies the statement as a Run XMap statement.

RunMapplet Statement

A RunMapplet statement calls a mapplet.

When you create a RunMapplet mapping statement, the Developer tool lists the mapplet reference objects associated with the transformation. Select the mapplet to call. The Developer tool creates an XMap statement with the mapplet name in the **Statement Type** field.

The input and output ports in the called mapplet must be the same type as the values being passed to it from the calling XMap. You can call a mapplet to perform tasks such as data masking, data quality, data lookup, and other activities usually related to relational transformation, without the necessity to convert data to relational format and then back to hierarchical format.

Note: The RunMapplet action can be used to call passive mapplets only.

RunMapplet Statement Properties

The RunMapplet statement contains properties that you can configure to customize the statement. You can configure the input, output, and a condition for executing the RunMapplet statement. The RunMapplet statement can contain the MappletInput statement and the MappletOutput statement.

The RunMapplet statement has the following properties:

Condition

Optional. An XPath expression that defines a condition for executing the RunMapplet statement. 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.

Input

Optional. 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.

Mode

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

- Add. Creates an element in the output hierarchical document. If the element in not multiple-occurring, and the same values exists in the output, the mapping statement fails.

- Match. The statement expects to find a match for the element in the output elements. The statement fails if the element does not exist in the output hierarchical document.
- Match or Add. If a matching element exists in the output hierarchical document, the Data Processor transformation does not add an output element. If the element does not exist in the output hierarchical document, the transformation creates an output element.

Name

Optional. 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.

On Fail

Optional. Determines the action taken if the statement fails. Choose one of the following options:

- Skip. If the statement fails, skip the statement.
- Propagate. If the statement fails, force the parent statement to also fail.

Output

Required. An XPath expression that defines the value of the node in the output hierarchy based on the results of the Input XPath expression.

Skip Missing Input

Optional. Determines whether to skip the statement if there is no match for the Input value. Choose one of the following options:

- Enabled. If the element is not in the input hierarchical document, the Data Processor transformation skips the statement without error.
- Disabled. The statement fails when the element is not in the input hierarchical document.

Statement Type

Required. Identifies the statement as a RunMapplet statement. The field identifies the name of the referenced mapplet.

MappletInput Statement

The MappletInput statement contains properties that you can configure to customize the statement. You can configure the input to map an input element to the mapplet input port. One or more MappletInput statements can be nested under the RunMapplet statement.

The MappletInput statement is performed to provide a value to be passed to the mapplet input. The values in the RunMapplet statement are passed to the mapplet ports in the same order that they are listed in the RunMapplet statement. If a statement is skipped, a null value is passed to the mapplet input port.

A MappletInput statement passes a single value. When the RunMapplet executes, the values from the nested MappletInput statements are collected and passed to the mapplet in the same order as the MappletInput statements.

MappletInput Statement Properties

The MappletInput statement contains properties that you can configure to customize the statement.

The RunMapplet statement has the following properties:

Default

Optional. The default value to use when an element is missing from the mapplet port input.

Input

Required. An XPath expression that evaluates to a sequence of nodes or values. The values are passed to the mapplet input ports.

Name

Optional. 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.

Skip Missing Input

Optional. Determines whether to skip the statement if there is no mapplet input port match for the input value. Choose one of the following options:

- Enabled. If the element is not in the input hierarchical document, the Data Processor transformation skips the statement without error.
- Disabled. The statement fails when the element is not in the input hierarchical document.

MappletOutput Statement

The MappletOutput statement is performed to obtain a value passed from the mapplet output. The values are passed from the mapplet ports to the RunMapplet output ports in the same order that they are listed in the RunMapplet statement. One or more MappletOutput statements can be nested under the RunMapplet statement.

A MappletOutput statement obtains a single value. When the RunMapplet executes, the values from the mapplet are collected to the nested MappletOutput statements in the same order as the MappletOutput statements.

MappletOutput Statement Properties

The MappletOutput statement contains properties that you can configure to customize the statement.

The RunMapplet statement has the following properties:

Default

Optional. The default value to use when testing the Data Processor transformation without actual mapping input. The default value is not used when there is mapping input.

Mode

Optional. Determines whether the transformation adds an output element or matches an existing element with a value from a mapping statement. Choose one of the following options:

- Add. Creates an element in the output hierarchical document. If the element in not multiple-occurring, and the same values exists in the output, the mapping statement fails.
- Match. The statement expects to find a match for the element in the output elements. The statement fails if the element does not exist in the output hierarchical document.
- Match or Add. If a matching element exists in the output hierarchical document, the Data Processor transformation does not add an output element. If the element does not exist in the output hierarchical document, the transformation creates an output element.

Name

Optional. 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.

Output

Required. An XPath expression that evaluates to a sequence of nodes or values. The mapplet output port values are passed to the sequence of nodes.

Creating an XMap

To create a Data Processor XMap object, choose the input and output schemas and add mapping statements.

1. On the Data Processor transformation **Objects** view create an XMap. Select an input schema, an example source, and an output schema.
2. To open the XMap editor, click the XMap object.
3. To create a Map, Group, or Repeating Group mapping statement, in the XMap editor drag and drop from a node in the input hierarchical schema to a node in the output hierarchical schema.

   The XMap editor creates a map link between the nodes. The mapping statement appears in the grid. The XMap editor automatically completes the mapping statement fields.
4. To create conditional logic in the grid, add a Router mapping statement as follows:
   a. Under the Router mapping statement, create Option mapping statements. Drag and drop input and output schema nodes into the Option statement fields in the grid.
   b. Under the Router mapping statement create one Default mapping statement to specify what happens if no Option mapping statement applies.
   c. Under the Option mapping statements, create Map mapping statements to specify conditions to map the input node to the output node.
5. To provide a common context for a group of statements, add a Group mapping statement. Nest Map mapping statements under the Group mapping statement.
6. To call another XMap object, add a Run XMap statement.
7. To change the context and logic for a mapping statement, edit mapping statement properties as follows:
   a. Demote statements to be child statements, or promote statements to be parent statements.
   b. Create XPath expressions to change the context or add predicates using the XPath editor.

Using the XMap Editor 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 a validation error. 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.

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.

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

1. To create a mapping statement, 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. Type a name for the mapping statement.
4. To define the input for the mapping statement, drag an element from the input schema to the Input field. Or, you can configure an XPath expression or a constant in the Input field.
5. To select the output node for the mapping statement, drag an element from the output schema to the Output field. You can also configure an XPath expression in the Output field.
6. To create an XPath expression with the XPath Expression editor for an Input, Output, or Condition field, click the **Open** button in the field.
7. To change the mapping statement type, click the **Open** button in the **Statement Type** field. Choose the mapping statement type from the list.

Mapping Statements Grid Interface

Edit mapping statements in the mapping statement grid in the XMap editor. You must create a statement before you can modify fields in the mapping statement grid.

You can perform the following tasks in the mapping statement grid:

- Drag nodes from the input or output schema to fields in the mapping statements.
- Copy elements into the mapping statement fields, or you can type values into the fields.
- You can move a mapping statement up or down in the Grid. Select the row and click the **Up** or **Down** arrow options.
- You can click **Demote** to indent a mapping statement under another statement. Click **Promote** if you want to move a statement up in the hierarchy.
- Click **Go to Row Number** to navigate to a row. Enter the row to navigate to. The Developer tool highlights the row for you.

Linking Schema Nodes

You can link an input schema node to an output schema node by dragging with the mouse. You can drag and drop to map a simple node to a simple node, a simple node to a complex node, a complex node to a simple node, or a complex node to a complex node.

The XMap editor creates a map link between the input schema node to the output schema node. The XMap editor also creates a mapping statement in the grid.

You can move drag and drop a mapping statement from one row in the grid to another row to change the XMap logic. For example, you can use this method to alter the sequence of Option statements within a Router.

Cut and Paste Mapping Statements

You can cut and paste mapping statements in the mapping statement grid in the XMap editor.

You can move statements up or down. You can paste mapping statements even within nested statements. You can promote a statement to be a parent statement, or demote a statement to be a child statement.

Pasted statements must usually be corrected as their logic is often out of context. After you paste a statement, you can modify fields in the mapping statement grid.

XPath Expressions

XPath expressions identify specific elements or nodes in hierarchical documents or check for conditions in the data. Use XPath expressions to define the Input, Condition, or Output fields of a mapping statement.

XPath is a syntax that defines parts of an hierarchical document. Use XPath to select sequences of nodes or values in an hierarchical 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:

XPath Expression	Description
nodename	Selects all child nodes of the given name in the context.
. (dot)	Selects current node.
..	Selects parent of current node.
@	Selects attribute.
/	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.
//	Selects nodes anywhere in the document or descendants of current node if preceded by node.

The following table lists some XPath expressions and the result of each expression:

XPath Expression	Result
/bookstore	Selects the root bookstore node.
bookstore/book	Selects book nodes that are children of all bookstore nodes.
//book	Selects the book nodes in the document in all locations.
bookstore//book	Selects all book nodes that are descendants of the bookstore nodes.

XPath Expression	Result
/bookstore/*	Selects all child nodes of bookstore root element.
//*	Returns a sequence of all elements in the document.

Predicates

A predicate is an expression that you configure to find a node in a hierarchical document. You can configure the expression to find a specific value. Create a predicate in an Input, 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 children 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 children 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 in the Output field when you configure a mapping statement with the Add mode or the Match or Add mode.

XPath Predicates Reference

XPath predicates find a matching node or sequence of nodes in a hierarchical document. A predicate expression defines the value of an Input, Condition, or Output field of a mapping statement. The XPath expression determines the context in which a mapping statement is run.

Use the following XPath expressions in predicates to select a node or sequence of nodes:

ancestor

Selects all ancestors, such as parent or grandparent, of the current node. For example, the predicate expression "ancestor::book" selects all book ancestors of the current node.

ancestor-or-self

Selects all ancestors, such as parent or grandparent, of the current node, as well as the current node. For example, the predicate expression "ancestor-or-self::book" selects all book ancestors of the current node, and the current node also.

attribute

Selects all attributes of the current node. For example, the predicate expression "attribute::lang" selects the lang attribute of the current node.

child

Selects all children of the current node. For example, the predicate expression "child::book" selects all book nodes that are children of the current node.

descendant

Selects all descendants, such as children or grandchildren, of the current node. For example, the predicate expression "descendant::book" selects all book descendants of the current node.

descendant-or-self

Selects all descendants, such as children or grandchildren, of the current node, as well as the current node. For example, the predicate expression "descendant-or-self::book" selects all book descendants of the current node, and the current node as well if it is a book node.

following

Selects everything in the document after the closing tag of the current node. For example, the predicate expression "following::book" selects everything in the document after the closing tag of the book node.

following-sibling

Selects all siblings following after the current node. For example, the predicate expression "following-sibling::book" selects all the siblings in the document after the book node.

namespace

Selects all namespace nodes of the current node.

parent

Selects the parent of the current node. For example, the predicate expression "parent::book" selects the lang attribute of the current node.

preceding

Selects all nodes that appear before the current node in the document, except ancestors, attribute nodes and namespace nodes. For example, the predicate expression "preceding::book" selects the nodes before the book node.

preceding-sibling

Selects all siblings that appear before the current node in the document. For example, the predicate expression "preceding-sibling::book" selects the sibling nodes before the book node.

self

Selects the current node. For example, the predicate expression "self::book" selects the current book node.

XPath Arithmetic Operators

To perform calculations, add arithmetic operators that evaluate hierarchical document nodes. You can add arithmetic operators to XPath expressions in the Input, Condition, or Output fields of a mapping statement.

The following table describes XPath arithmetic operators that you can use in XMap expressions:

XPath Expression	Description
|	Selects two node sets in context. For example, the predicate expression "//book | //cd" returns a node set with all book and cd elements.
+	Adds the elements. For example, the predicate expression "1+2" returns 3.
-	Subtracts the elements. For example, the predicate expression "2-1" returns 1.

XPath Expression	Description
*	Multiplies the elements. For example, the predicate expression "2*1" returns 2.
div	Divides the elements. For example, the predicate expression "6 div 3" returns 2.
=	Selects the elements that equal the expression. For example, the predicate expression "cost=1.50" returns true if the cost is 1.50, and false if the cost is 1.60.
!=	Selects the elements that are not equal to the expression. For example, the predicate expression "cost!=1.50" returns true if the cost is 1.60, and false if the cost is 1.50.
<	Selects the elements that are less than the expression. For example, the predicate expression "tax<1.50" returns true if the tax is 1.00, and false if the tax is 1.50.
<=	Selects the elements that are equal to or less than the expression. For example, the predicate expression "tax<=1.50" returns true if the tax is 1.50, and false if the tax is 1.80.
>	Selects the elements that are greater than the expression. For example, the predicate expression "tax>1.50" returns true if the tax is 1.90, and false if the tax is 1.50.
>=	Selects the elements that are equal to or greater than the expression. For example, the predicate expression "tax>=1.50" returns true if the tax is 1.50, and false if the tax is 1.00.
or	Selects the elements that can satisfy one or more conditions. For example, the predicate expression "tax=1.50 or tax=1.70" returns true if the tax is 1.50, and false if the tax is 1.00.
and	Selects the elements that can satisfy all the given conditions. For example, the predicate expression "price>1.00 and price<1.90" returns true if the price is 1.50, and false if the price is 1.00.
mode	Performs division and provides the remainder. For example, the predicate expression "3 mod 2" returns 1.

Output XPath Expressions

The Data Processor transformation accepts a subset of XPath statements in the Output field when the Mode field set to is **Add** or **Match or Add**. When you select these mode settings, the Data Processor transformation creates elements as needed to match the XPath expression in the Output field.

You can use a simple XPath expression in the Output field. A simple expression has child axes, parent axes, or variables. Simple expressions do not have predicates, functions or complex axes. For example, you can use the following Output field expressions:

```
person/data
```

```
/root/ceo/name
```

```
$var/name
```

```
person/../ceo
```

You can use a simple predicate with cardinality for an element with several instances. For example, you can use the following Output field expression:

```
person/phone[4]
```

You can use a simple predicate with a formula with an equal sign, with simple XPaths on the left-hand side of the equal sign. For example, you can use the following Output field expressions:

```
Person[id=10]

Person[id=$id]

Person[id=@dp:input()/ID]

Company[name=upper-case($compName)]

Person[role="manager" and id=1]
```

You can also use a combination of a simple expression with cardinality and a formula that uses a simple XPath on the left-hand side of the equal sign. For example, you can use the following Output field expressions:

```
company[4]/details[id=$myid]/phone
```

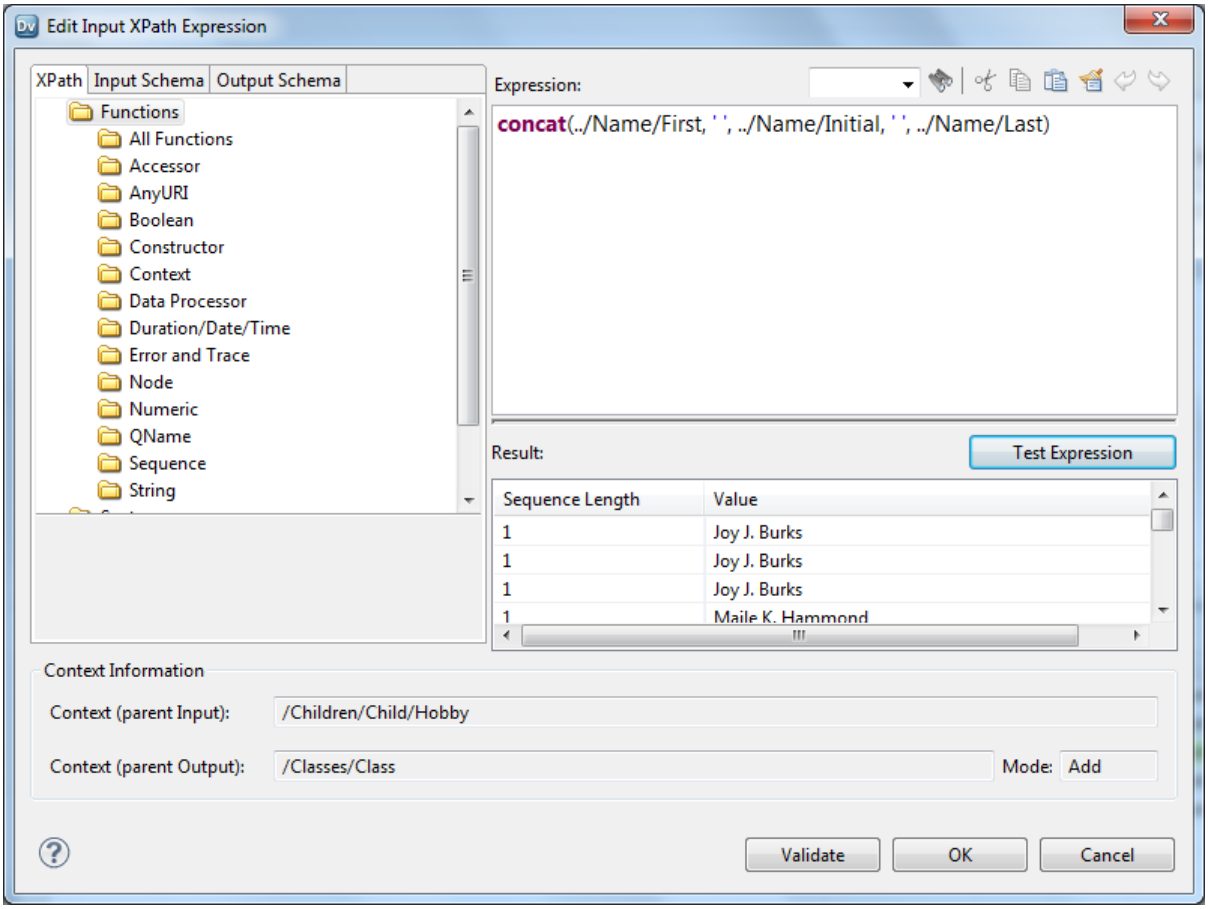
Note: When the Mode field is set to **Match**, the Output field can also accept complex XPath expressions.

XPath Expression Editor

Create expressions in the XPath Expression Editor. XPath is a query language used to select nodes in an hierarchical document and perform computations.

You can use XPath expressions to define the context for a mapping statement. You can define conditions to transform and filter the data using different XPath expressions in mapping statement properties. You can add various arithmetic calculations to a mapping statement using XPath expressions.

When you click the Open button in the Input, Condition, or Output field, the Expression Editor appears. The following figure shows the XPath Expression Editor:



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 Data Processor transformation uses the following XPath functions:

dp:as_xml

Receives a node as an input and returns the node value and the value of all the children as an XML string recursively. The as_xml function uses the following syntax: `dp:as_XML(<node>)`

dp:get_id

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 schema and map it to keys in relational data. The get_id function uses the following syntax: `dp:get_id(<node>)`

dp:input

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. The input function uses the following syntax: `dp:input()`

dp:transform

Call a Data Processor transformation transformer that you define in a Script. You can perform an inline or external transformation. The function uses the following syntax: `dp:transform(<transform-name>,<transform-value>)`

The transform function uses the following parameters:

- Transform-name. The name of the transformer in the Script.
- Transform-value. The transform value upon which to perform the transformation.

Note: The lookup function is available through the use of the **dp:transform** function.

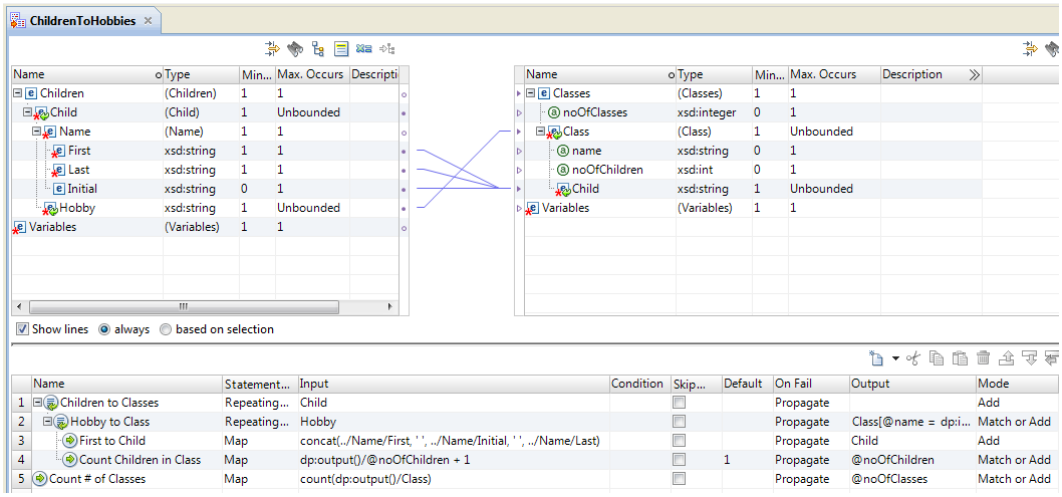
dp:output

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. The output function uses the following syntax: `dp:output()`

XPath Expressions Example

Use XPath expressions in mapping statements. XPath expressions identify the elements in the input document to be mapped and transformed in the output document. An XPath expression is also used to perform an arithmetic operation.

The following figure shows XPath expressions in the grid:



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 chldren in each class. Another statement counts the number of classes.

The XMap contains the following expressions:

Grid row 2 expression <Class[@name = dp:input()]>

Adds a Class element or find a Class that matches Hobby. The dp:input() is required because the expression refers to an input element.

Grid row 3 expression <concat(../Name/First,' ',../Name/Initial,' ',../Name/Last)>

Concatenates the first name, middle initial, and last name and adds spaces between them.

Grid row 4 expression <dp:output()/@noOfChildren + 1>

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.

Grid row 5 expression <count(dp:output()/Class)>

Counts the Class elements. The dp:output() function is required because the expression refers to an output element.

Creating An Expression

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. To add an element to an expression, double-click elements in the **Navigation** panel.
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, it 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 Schema Example

The XML Input schema for the XMap example has the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2012 sp1 (x64) (http://www.altova.com) by Informatica
Corporation (Informatica Corporation) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="company"
targetNamespace="company" elementFormDefault="qualified"
attributeFormDefault="unqualified">

<xs:element name="Employee" type="EmployeeType"/>

<xs:element name="Input">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="Company" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="Company">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
            <xs:element ref="Department" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="Department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
            <xs:element ref="Employee" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>


<xs:complexType name="EmployeeType">
    <xs:sequence>
        <xs:element name="FirstName" type="xs:string" minOccurs="0"/>
        <xs:element name="LastName" type="xs:string" minOccurs="0"/>
        <xs:element name="Role" type="xs:string" minOccurs="0"/>
        <xs:element name="StartDate" type="xs:date" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string"/>
</xs:complexType>

</xs:schema>
```

The schema root is Input. Input has multiple-occurring Companies. With in each Company are multiple Departments. Each Department has Employees. Employee has a Role that determines whether the employee is a manager or not.

XML Output Schema Example

The XML Output schema for the XMap example has the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2012 sp1 (x64) (http://www.altova.com) by Informatica
Corporation (Informatica Corporation) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="organization"
targetNamespace="organization" elementFormDefault="qualified"
attributeFormDefault="unqualified">

<xs:element name="Worker" type="WorkerType"/>

<xs:element name="Output">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="Organization" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>


<xs:element name="Organization">
    <xs:complexType>
        <xs:sequence>
            <xs:choice maxOccurs="unbounded">
                <xs:element name="Worker" type="WorkerType"/>
                <xs:element name="Manager" type="WorkerType"/>
            </xs:choice>
            <xs:element name="Department" type="xs:string" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="noOfEmployees" type="xs:int"/>
    </xs:complexType>
</xs:element>

<xs:complexType name="WorkerType">
    <xs:sequence>
        <xs:element name="FirstName" type="xs:string" minOccurs="0"/>
        <xs:element name="LastName" type="xs:string" minOccurs="0"/>
        <xs:element name="FullName" type="xs:string" minOccurs="0"/>
        <xs:element name="Id" type="xs:string"/>
        <xs:element name="YearsOfService" type="xs:int" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>

</xs:schema>
```

The schema root is Output. Output has multiple-occurring Organizations. Within each Organization are Workers and Managers. Workers and Managers are WorkerTypes. A Worker and a Manager contain the same elements. The WorkerType includes a YearsOfService element that the XMap calculates based on the StartDate.

XML Input Data

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

```
<?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>
```

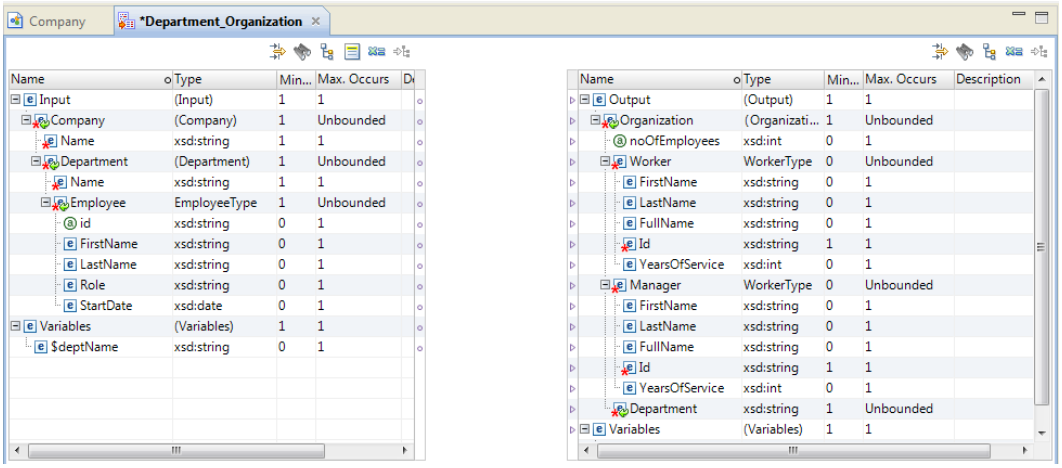
```
                <Role>Manager</Role>
                <StartDate>1993-04-21</StartDate>
            </Employee>

            <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>

            <Employee id="289021615">
                <FirstName>Christen</FirstName>
                <LastName>Fulton</LastName>
                <Role>Worker</Role>
                <StartDate>1997-11-16</StartDate>
            </Employee>

            <Employee id="761338290">
                <FirstName>Felix</FirstName>
                <LastName>Boyd</LastName>
                <Role>Worker</Role>
                <StartDate>2009-12-29</StartDate>
            </Employee>

        </Department>
    </Company>
```

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.

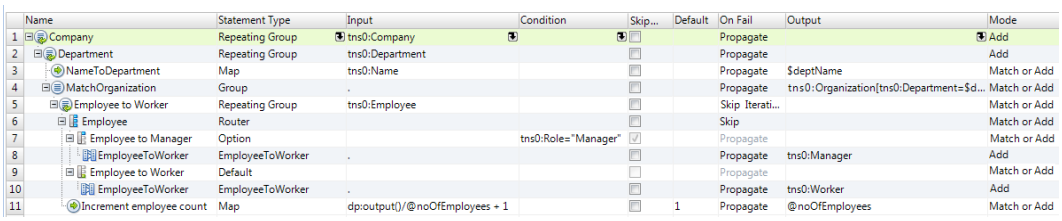
The following figure shows the input and output XML hierarchies:



Mapping Statements in the Example

Use the grid area of the XMap editor to define statements that map the input XML elements to the output XML elements. Create and edit mapping statements in the grid. Define the context, condition and expected input and output for a mapping statement. You can add variables to mapping statements.

The following figure shows the mapping statements in the grid:



	Name	Statement Type	Input	Condition	Skip...	Default	On Fail	Output	Mode
1	Company	Repeating Group	tns0:Company				Propagate		Add
2	Department	Repeating Group	tns0:Department				Propagate		Add
3	NameToDepartment	Map	tns0:Name				Propagate	$deptName	Match or Add
4	MatchOrganization	Group	.				Propagate	tns0:Organization[tns0:Department=$d...	Match or Add
5	Employee to Worker	Repeating Group	tns0:Employee				Skip Iterati...		Match or Add
6	Employee	Router					Skip		Match or Add
7	Employee to Manager	Option		tns0:Role="Manager"			Propagate		Match or Add
8	EmployeeToWorker	EmployeeToWorker	.				Propagate	tns0:Manager	Add
9	Employee to Worker	Default					Propagate		Match or Add
10	EmployeeToWorker	EmployeeToWorker	.				Propagate	tns0:Worker	Add
11	Increment employee count	Map	dp:output()/@noOfEmployees + 1			1	Propagate	@noOfEmployees	Match or Add

The grid contains the following mapping statements:

Grid row 1, Repeating Group statement named Company

The Company statement is a Repeating Group statement. It repeats for each the 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.

Grid row 2, Repeating Group statement named Department

The Department statement is a Repeating Group statement. It 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.

Grid row 3, Map statement named NametoDepartment

The NametoDepartment statement is a Map statement. It maps the Name element to a variable $deptName.

Grid row 4, Repeating Group statement named MatchOrganization

The MatchOrganization statement is a Repeating Group statement. It has an output expression:

```
tns0:Organization[tns0:Department=$deptName]
```

The statement finds the Organization element in the output that contains a Department child element with the value in $deptName. Or, if the Department element does not exist, the element is created.

Grid row 5, Repeating Group statement named EmployeeToWorker

The EmployeeToWorker statement is a Repeating Group statement. It repeats for each Employee element.

Grid row 6, Router statement named Employee

The Employee statement is a Router statement. The statement has no input or output.

Grid row 7, Option statement named EmployeeToMgr

The EmployeeToMgr statement is an Option statement. The Option statement contains the following condition:

```
tns0:Role="Manager".
```

When the Role is Manager, the statement is true, and the Data Processor transformation evaluates the statements nested inside the Option statement.

Grid row 8, Run XMap statement named EmployeeToWorker

The EmployeeToWorker statement is a Run XMap statement. It calls the XMap_EmployeesToRoles XMap to pass the Employee elements to the Manager type.

Grid row 9, Default statement named EmployeeToWorker

The EmployeeToWorker statement is a Default statement. The Data Processor transformation performs the child statements when the employee role is not a manager.

Grid row 10, Run XMap statement named EmployeeToWorker

The EmployeeToWorker statement is a Run XMap statement. It calls the XMap_EmployeesToRoles XMap to pass the Employee elements to the Worker type.

Grid row 11, Map statement named IncrementEmployeeCount

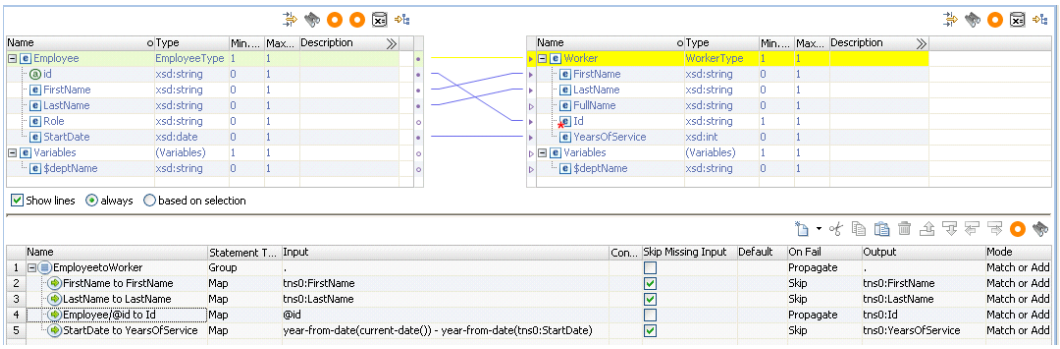
The IncrementEmployeeCount statement is a Map statement. It calls the Data Processor transformation to add 1 to @noOfEmployees for each Employee that it iterates. The Map statement contains the following input expression:

```
dp:output()/@ noOfEmployees + 1.
```

Group Statements Example

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:



The grid contains the following mapping statements:

Grid row 1, Group statement named EmployeeToWorker

The EmployeeToWorker statement is a Group statement. It provides context for the rest of the mapping statements.

Grid row 2, Map statement named FirstNametoFirstName

The FirstNametoFirstName statement is a Map statement. It maps the first name to the first name.

Grid row 3, Map statement named LastNametoLastName

The LastNametoLastName statement is a Map statement. It maps the last name to the last name.

Grid row 4, Map statement named Employee/@IDtoID

The Employee/@IDtoID statement is a Map statement. It maps the employee ID to the employee ID.

Grid row 5, Map statement named StartDatetoYearsofService

The StartDatetoYearsofService statement is a Map statement. It determines the number of years of service by subtracting a start-date from the current-date.

CHAPTER 7

Libraries

This chapter includes the following topics:

- Libraries Overview, 120
- Library Structure, 121
- Element Properties, 121
- Library Management, 121
- Edit Libraries with the Library Editor, 122
- Edit Libraries with the IntelliScript Editor, 124

Libraries Overview

A Library is a Data Processor transformation object that contains predefined components used to transform a range of industry messaging standards. A Data Processor transformation uses a Library to transform an industry message type input into other formats. You can create Library objects for all libraries.

A Library contains a large number of objects and components, such as Parsers, Serializers, and XML schemas, that transform the industry standard input and specific application messages into XML output. A Library might contain objects for message validation, acknowledgments, and diagnostic displays. A Library uses objects to transform the messaging type from industry standard input to XML and from XML to other formats.

You can create Library objects for ACORD, BAI, CREST, DTCC-NSCC, EDIFACT, EDIT-UCS & WINS, EDI_VICS, EDI-X12, FIX, FpML, HIPAA, HIX, HL7, IATA, IDS, MDM Mapping, NACHA, NCPDP, SEPA, SWIFT, and Telekurs libraries.

You can use a dedicated Library editor to edit the Library specifications for the DTCC-NSCC, EDIFACT, EDI-X12, HIPAA, HL7, and SWIFT libraries. A Library message contains a root element, container elements, and data elements. The types of container and data elements vary according to message type. You can add and delete elements and configure the properties of elements to change validation settings.

For more information about industry message types, see *Data Transformation Libraries Guide*.

Library Structure

A Library is a set of transformations. All Libraries contain Parsers, Serializers, and XML schemas. Some Libraries contain additional components for message validation, acknowledgments, and diagnostic displays. A Library object is a set of components that convert a specific industry message type.

When you create a Data Processor transformation, you can include a Library object instead of creating your own Scripts to transform a standard industry message type. You can use a Library object without modification, or you can edit it based on your requirements.

Each Library object transforms a particular industry standard. For example, the HL7 Library contains components for each of the message types or data structures available in the HL7 messaging standard for medical information systems.

The Library contains specific types of message types. For example, the HL7 Library contains messages such as:

```
ADT_A01_Admit_a_Patient
ADT_A02_Transfer_a_Patient
```

Element Properties

A Library message contains a root element, container elements, and data elements. The types of container and data elements vary according to message type. You can configure both container element and data element properties with the Library editor.

Properties have the following categories:

Global Settings

Properties that have the same value for an element with more than one instance, each occupying a different position in the hierarchy. Any change to a Global Settings property appears in all instances of the element.

Positional Settings

Properties whose value can differ for each instance of the element in the hierarchy.

Element properties are named in accordance to industry standards. The Library editor displays different properties for different Libraries.

Library Management

You can use the Library editor to customize the message type structures and elements for the DTCC-NSCC, EDIFACT, EDI-X12, HIPAA, HL7, and SWIFT libraries. To change the structure of the message type, use the Library editor to add, edit and delete elements from the Library message.

You might want to change the way that a message type is transformed. A Library transformation contains a large number of objects and components, such as Scripts with Parsers, Serializers, and XML schemas, that define the transformation. A Library transformation might contain objects for message validation, acknowledgments, and diagnostic displays.

A Library transformation uses its objects to transform the messaging type from industry standard input to XML and from XML to other formats. To access and edit the Scripts, XMaps and schemas that you created with the Library editor, you must generate the Library objects. Generate the Library objects only if you need to make a change to the Library objects that you cannot do with the Library editor.

After you generate the Library objects, use the IntelliScript editor to edit Parsers, Serializers, and Mappers. For example, you want to change the output structure to suit your requirements. You generate the Library objects and edit the Scripts with the IntelliScript editor.

When you create a Library for library packages that do not support the Library editor, the Library objects are pre-generated. You do not need to generate the Library objects. You can use the IntelliScript editor to directly edit the Script components for these libraries.

After you generate Library objects, or if the Library objects were pre-generated, you cannot edit the Library elements with the Library editor. To use the Library editor again, you must discard the generated Library objects for those libraries that you can edit with the Library editor. For example, you might decide that you want to add input fields but not change the structure of the output. Discard the Library objects and after that add custom elements with the Library editor.

Note: Any changes that you made to the generated Library objects are lost when you discard the generated objects.

Library Example for EDI-X12

You and your suppliers use the X12 850 Purchase Order standard to document and process orders. You want to change the library message type specifications from the industry standards to support your internal processes.

The furniture manufacturing business needs standard elements such as model, color, and size, and a custom element for the fabric type. It does not need a batch number or expiration date. Use the Library editor to remove elements and add custom elements.

You can also use the Library editor to change element properties, add segments, and copy elements. For example, change the element property which defines the list of colors to add new colors. To add a manufacturer code element to the fabric segment and hardware segment, you can copy it.

You run a Data Processor transformation that contains your customized Library. The text input from your suppliers is formatted in a modified version of the X12 850 Purchase Order standard. The modified input format matches the changes that you made to the Library. The text is transformed into hierarchical output that can be sent to other transformations for further processing.

Edit Libraries with the Library Editor

You can use a dedicated Library editor to edit the Library specifications for the EDI-X12, SWIFT, HL7, HIPAA, DTCC-NSCC, and EDIFACT libraries.

The Library editor lets you customize the message type structures and elements with an editor that is customized for each message type. You can add, edit and delete elements from the Library object with the Library Editor.

Adding an Element with the Library Editor

A Library message contains a root element, container elements, and data elements. Use the Library editor to add an element to a message.

1. In the **Objects** view, select the Library editor object and click **Open**.

   The Library editor appears.
2. Click the **Add element** icon and choose where you want to add the element.
   - **New**. Append the element to the end of the list of elements.
   - **Insert Above**. Insert the element above the element selected in the Library editor.
   - **Insert Below**. Insert the element below the element selected in the Library editor.
   - **Insert Within**. Insert the element within the container element selected in the Library editor.
3. Select whether to copy an existing element or create a new element and click **OK**.

Editing the Element Properties with the Library Editor

Use the Library editor to edit a container element or data element in a Library. When you change the properties of an element, you change the specifications for the message type.

1. In the **Message Definition** window of the Library editor, select the element to edit.

   The **Properties** window displays the element properties.
2. In the **Properties** window, select a property and enter the new property value.

   If you enter an incorrect value type or an out-of-range value, you will be prompted to correct the value.

Testing a Library

Test the Library object in the **Data Viewer** view.

Before you test the Library, select a sample input file to test.

1. To select a sample input file, in the Library editor **General** panel, near the **Sample input** field, browse to select a sample input file.
2. Open the **Data Viewer** view.
3. To select the Library message type that you edited as the component to run, click **Synchronize with Editor**.
4. Click **Run**.

   The Developer tool runs the Library object Parser. The output results appear in the **Output** window.
5. If there are validation errors, the **Output Errors** panel in the **Output** window lists the errors and their location. To find the source of the error, double-click the error.
6. To view the validation error files, click the file names in the **Additional Outputs** panel in the **Output** window.

   When you click the `ErrorsFound.txt` or `Errors.xml` file name, the file opens in an external browser.

Generating the Library Objects

Generate the Library objects so that you can access them directly with the Data Processor transformation editors. After you generate the Library objects, you cannot use the Library editor to edit the Library.

You must generate Library objects to access and edit the pre-configured parsers, mappers, serializers, and XML schemas associated with the Library.

1. In the **Objects** view, right-click the Library and select **Generate Library Objects.**
2. To access the Library objects, select **Yes**.

   The Developer tool creates a **Generated Library Objects** folder that contains the Library objects. The startup component is generated.

   Note: If you want to change which Library objects are generated, select a different component.
3. To edit a Script or schema, select it and click **Open**.

   Use the IntelliScript editor to edit the Script.

Discarding the Library Objects

To edit the Library elements with the Library editor, you must remove the Library objects that you generated. When you discard the Library objects, any change that you made to the them is lost.

1. In the **Objects** view, right-click the Library and select **Discard Generate Library Objects.**
2. To discard the Library objects from the Data Processor transformation editors, select **Yes**.

   The **Generated Library Objects** folder that is discarded. The Library components and objects still exist, but are not accessible through the Data Processor transformation editors.

Edit Libraries with the IntelliScript Editor

When you create a Library for the ACORD, BAI, FIX, HIX, IATA, IDS, NACHA, NCPDP, Telekurs, Bloomberg, SEPA, FpML, and Thompson Reuters libraries, you can use the IntelliScript editor to directly edit the Script components for these libraries. You do not need to generate the Library objects.

For the EDI-X12, SWIFT, HL7, HIPAA, DTCC-NSCC, Edifact, SEPA, and FpML libraries, you must generate the Library objects to access and edit the Scripts, XMaps, and schemas that you created with the Library editor with the IntelliScript editor.

The IntelliScript editor is a graphical tool that you use to edit Scripts. Use the IntelliScript editor to add Script components to the Script and configure Script component properties. For more information about Scripts and the IntelliScript editor, see "Scripts Overview" on page 140.

If you imported a Library project from a previous version as a Data Transformation service, you can edit the Script components with the IntelliScript editor.

CHAPTER 8

Schema Object

This chapter includes the following topics:

- Schema Object Overview, 125
- Schema Object Overview View, 125
- Schema Object Schema View, 126
- Schema Object Advanced View, 132
- Creating a Schema Object, 132
- Schema Updates, 133

Schema Object Overview

A schema object is a hierarchical schema that you import to the Model repository. After you import the schema, you can view the schema components in the Developer tool. You can import an Avro, Parquet, XML, or JSON schema. The Developer tool converts the schema to an .xsd file in the Model repository.

When you create a SOAP web service, you can define the structure of the web service based on a hierarchical schema. When you create a web service without a WSDL, you can define the operations, the input, the output, and the fault signatures based on the types and elements that the schema defines.

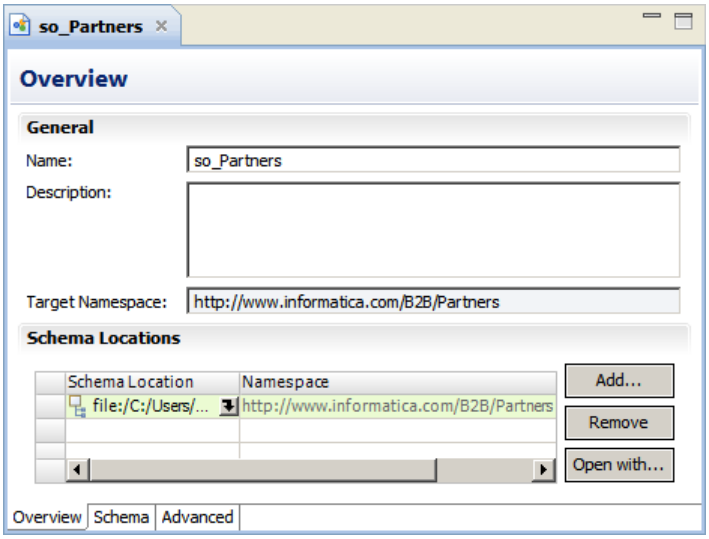
When you import a schema you can edit general schema properties in the **Overview** view. Edit advanced properties in the **Advanced** view. View the schema file content in the **Schema** view.

Schema Object Overview View

Select the **Overview** view to update the schema name or schema description, view namespaces, and manage schema files.

The **Overview** view shows the name, description, and target namespace for the schema. You can edit the schema name and the description. The target namespace displays the namespace to which the schema components belong. If no target namespace appears, the schema components do not belong to a namespace.

The following figure shows the **Overview** view of a schema object:



The **Schema Locations** area lists the schema files and the namespaces. You can add multiple root .xsd files. If a schema file includes or imports other schema files, the Developer tool includes the child .xsd files in the schema.

Schema Files

You can add multiple root-level .xsd files to a schema object. You can also remove root-level .xsd files from a schema object.

When you add a schema file, the Developer tool imports all .xsd files that are imported by or included in the file you add. The Developer tool validates the files you add against the files that are part of the schema object. The Developer tool does not allow you to add a file if the file conflicts with a file that is part of the schema object.

For example, a schema object contains root schema file "BostonCust.xsd." You want to add root schema file "LACust.xsd" to the schema object. Both schema files have the same target namespace and define an element called "Customer." When you try to add schema file LACust.xsd to the schema object, the Developer tool prompts you to retain the BostonCust.xsd file or overwrite it with the LACust.xsd file.

You can remove any root-level schema file. If you remove a schema file, the Developer tool changes the element type of elements that were defined by the schema file to xs:string.

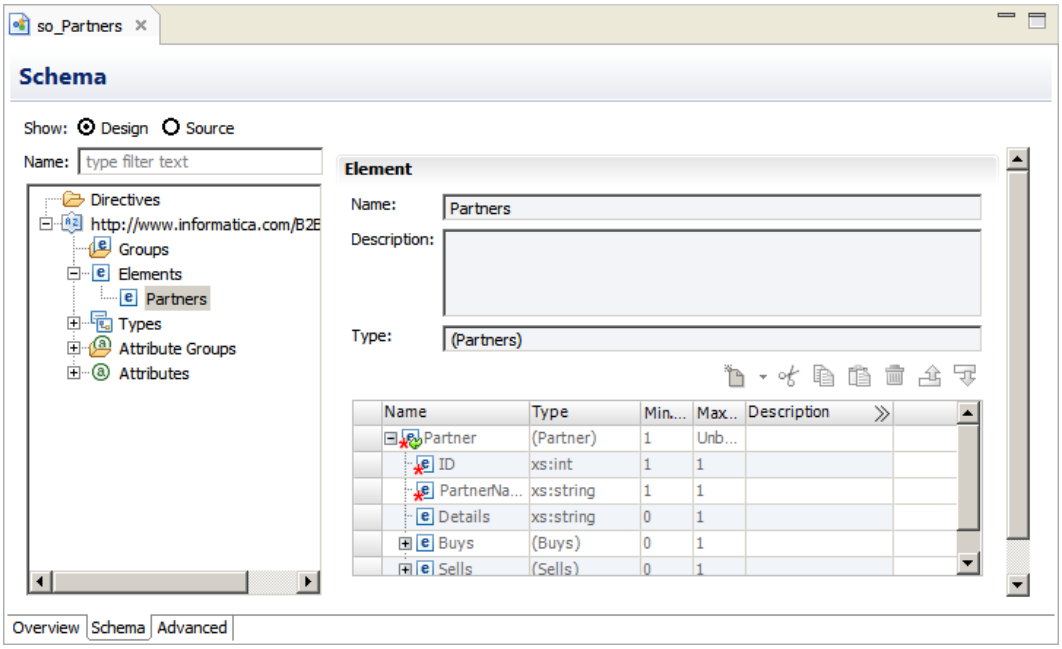
To add a schema file, select the **Overview** view, and click the **Add** button next to the **Schema Locations** list. Then, select the schema file. To remove a schema file, select the file and click the **Remove** button.

Schema Object Schema View

The **Schema** view shows an alphabetic list of the groups, elements, types, attribute groups, and attributes in the schema. When you select a group, element, type, attribute group, or attribute in the **Schema** view, properties display in the right panel. You can also view each .xsd file in the **Schema** view.

The **Schema** view provides a list of the namespaces and the .xsd files in the schema object.

The following figure shows the **Schema** view of a schema object:



You can perform the following actions in the **Schema** view:

- To view the list of schema constructs, expand the **Directives** folder. To view the namespace, prefix, and the location, select a schema construct from the list.
- To view the namespace prefix, generated prefix, and location, select a namespace. You can change the generated prefix.
- To view the schema object as an .xsd file, select **Source**. If the schema object includes other schemas, you can select which .xsd file to view.
- To view an alphabetic list of groups, elements, types, attribute groups, and attributes in each namespace of the schema, select **Design**. You can enter one or more characters in the **Name** field to filter the groups, elements, types, attribute groups, and attributes by name.
- To view the element properties, select a group, element, type, attribute group, or attribute. The Developer tool displays different fields in the right panel based on the object you select.

When you view types, you can see whether a type is derived from another type. The interface shows the parent type. The interface also shows whether the child element inherited values by restriction or extension.

Namespace Properties

The **Namespace** view shows the prefix and location for a selected namespace.

The namespace associated with each schema file differentiates between elements that come from different sources but have the same names. A Uniform Resource Identifier (URI) reference defines the location of the file that contains the elements and attribute names.

When you import a schema that contains more than one namespace, the Developer tool adds the namespaces to the schema object. When the schema file includes other schemas, the namespaces for those schemas are also included.

The Developer tool creates a generated prefix for each namespace. When the schema does not contain a prefix, the Developer tool generates the namespace prefix tns0 and increments the prefix number for each additional namespace prefix. The Developer tool reserves the namespace prefix xs. If you import a schema

that contains the namespace prefix xs, the Developer tool creates the generated prefix xs1. The Developer tool increments the prefix number when the schema contains the generated prefix value.

For example, Customer_Orders.xsd has a namespace. The schema includes another schema, Customers.xsd. The Customers schema has a different namespace. The Developer tool assigns prefix tns0 to the Customer_Orders namespace and prefix tns1 to the Customers namespace.

To view the namespace location and prefix, select a namespace in the **Schema** view.

When you create a web service from more than one schema object, each namespace must have a unique prefix. You can modify the generated prefix for each namespace.

Element Properties

An element is a simple or a complex type. A complex type contains other types. When you select an element in the **Schema** view, the Developer tool lists the child elements and the properties in the right panel of the screen.

The following table describes element properties that appear when you select an element:

Property	Description
Name	The element name.
Description	Description of the type.
Type	The element type.

The following table describes the child element properties that appear when you select an element:

Property	Description
Name	The element name.
Type	The element type.
Minimum Occurs	The minimum number of times that the element can occur at one point in an instance.
Maximum Occurs	The maximum number of times that the element can occur at one point in an instance.
Description	Description of the element.

To view additional child element properties, click the double arrow in the Description column to expand the window.

The following table describes the additional child element properties that appear when you expand the Description column:

Property	Description
Fixed Value	A specific value for an element that does not change.
Nillable	The element can have nil values. A nil element has element tags but has no value and no content.

Property	Description
Abstract	The element is an abstract type. An instance must include types derived from that type. An abstract type is not a valid type without derived element types.
Minimum Value	The minimum value for an element in an instance.
Maximum Value	The maximum value for an element in an instance.
Minimum Length	The minimum length of an element. Length is in bytes, characters, or items based on the element type.
Maximum Length	The maximum length of an element. Length is in bytes, characters, or items based on the element type.
Enumeration	A list of all legal values for an element.
Pattern	An expression pattern that defines valid element values.

Advanced Element Properties

To view advanced properties for a element, select the element in the **Schema** view. Click **Advanced**.

The following table describes the element advanced properties:

Property	Description
Abstract	The element is an abstract type. A SOAP message must include types derived from that type. An abstract type is not a valid type without derived element types.
Block	Prevents a derived element from appearing in the hierarchy in place of this element. The block value can contain "#all" or a list that includes extension, restriction, or substitution.
Final	Prevents the schema from extending or restricting the simple type as a derived type.
Substitution Group	The name of an element to substitute with the element.
Nillible	The element can have nil values. A nil element has element tags but has no value and no content.

Simple Type Properties

A simple type element is an element that contains unstructured text. When you select a simple type element in the **Schema** view, information about the simple type element appears in the right panel.

The following table describes the properties you can view for a simple type:

Property	Description
Type	Name of the element.
Description	Description of the element.

Property	Description
Variety	Defines if the simple type is union, list, anyType, or atomic. An atomic element contains no other elements or attributes.
Member types	A list of the types in a UNION construct.
Item type	The element type.
Base	The base type of an atomic element, such as integer or string.
Minimum Length	The minimum length for an element. Length is in bytes, characters, or items based on the element type.
Maximum Length	The maximum length for an element. Length is in bytes, characters, or items based on the element type.
Collapse whitespace	Strips leading and trailing whitespace. Collapses multiple spaces to a single space.
Enumerations	Restrict the type to the list of legal values.
Patterns	Restrict the type to values defined by a pattern expression.

Simple Type Advanced Properties

To view advanced properties for a simple type, select the simple type in the **Schema** view. Click **Advanced**.

The advanced properties appear below the simple type properties.

The following table describes the advanced property for a simple type:

Property	Description
Final	Prevents the schema from extending or restricting the simple type as a derived type.

Complex Type Properties

A complex type is an element that contains other elements and attributes. A complex type contains elements that are simple or complex types. When you select a complex type in the **Schema** view, the Developer tool lists the child elements and the child element properties in the right panel of the screen.

The following table describes complex type properties:

Property	Description
Name	The type name.
Description	Description of the type.
Inherit from	Name of the parent type.
Inherit by	Restriction or extension. A complex type is derived from a parent type. The complex type might reduce the elements or attributes of the parent. Or, it might add elements and attributes.

To view properties of each element in a complex type, click the double arrow in the Description column to expand the window.

Complex Type Advanced Properties

To view advanced properties for a complex type, select the element in the **Schema** view. Click **Advanced**.

The following table describes the advanced properties for a complex element or type:

Property	Description
Abstract	The element is an abstract type. A SOAP message must include types derived from that type. An abstract type is not a valid type without derived element types.
Block	Prevents a derived element from appearing in the schema in place of this element. The block value can contain "#all" or a list that includes extension, restriction, or substitution.
Final	Prevents the schema from extending or restricting the simple type as a derived type.
Substitution Group	The name of an element to substitute with the element.
Nillible	The element can have nil values. A nil element has element tags but has no value and no content.

Attribute Properties

An attribute is a simple type. Elements and complex types contain attributes. Global attributes appear as part of the schema. When you select a global attribute in the **Schema** view, the Developer tool lists attribute properties and related type properties in the right panel of the screen.

The following table describes the attribute properties:

Property	Description
Name	The attribute name.
Description	Description of the attribute.
Type	The attribute type.
Value	The value of the attribute type. Indicates whether the value of the attribute type is fixed or has a default value. If no value is defined, the property displays default=0.

The following table describes the type properties:

Property	Description
Minimum Length	The minimum length of the type. Length is in bytes, characters, or items based on the type.
Maximum Length	The maximum length of the type. Length is in bytes, characters, or items based on the type.
Collapse Whitespace	Strips leading and trailing whitespace. Collapses multiple spaces to a single space.

Property	Description
Enumerations	Restrict the type to the list of legal values.
Patterns	Restrict the type to values defined by a pattern expression.

Schema Object Advanced View

View advanced properties for the schema object.

The following table describes advanced properties for a schema object:

Name	Value	Description
elementFormDefault	Qualified or Unqualified	Determines whether or not elements must have a namespace. The schema qualifies elements with a prefix or by a target namespace declaration. The unqualified value means that the elements do not need a namespace.
attributeFormDefault	Qualified or Unqualified	Determines whether or not locally declared attributes must have a namespace. The schema qualifies attributes with a prefix or by a target namespace declaration. The unqualified value means that the attributes do not need a namespace.
File location	Full path to the .xsd file	The location of the .xsd file when you imported it.

Creating a Schema Object

You can import a hierarchical schema file or sample file to create a schema object in the repository.

1. Select a project or folder in the **Object Explorer** view.
2. Click **File** > **New** > **Schema**.

   The **New Schema** dialog box appears.
3. To import a schema file, select **Create from schema**, and then browse to and select a hierarchical schema file.

   You can enter a URI or a location on the file system to browse. The Developer tool validates the schema you choose. Review validation messages. You can select an Avro, Parquet, JSON, or .xsd schema file.

   Note: If the URI contains non-English characters, the import might fail. Copy the URI to the address bar in any browser. Copy the location back from the browser. The Developer tool accepts the encoded URI from the browser.
4. To create a schema from a sample file, select **Create from a sample file**, and then browse to and select a hierarchical file.

   You can select an Avro, Parquet, JSON, or XML file.

Note: If you select a file with a different extension that contains Avro, Parquet, JSON, or XML content, the wizard recognizes the file content.

5. Optionally, change the schema name.
6. Click **Next** to view a list of the elements and types in the schema.
7. Click **Finish** to import the schema.

   The schema appears under Schema Objects in the **Object Explorer** view. The Developer tool stores the schema as an .xsd file.
8. To change the generated prefix for a schema namespace, select the namespace in the **Object Explorer** view. Change the **Generated Prefix** property in the **Namespace** view.

Schema Updates

You can update a schema object when elements, attributes, types, or other schema components change. When you update a schema object, the Developer tool updates objects that use the schema.

You can update a schema object through the following methods:

Synchronize the schema.

Synchronize a schema object when you update the schema files outside the Developer tool. When you synchronize a schema object, the Developer tool reimports all of the schema .xsd files that contain changes.

Edit a schema file.

Edit a schema file when you want to update a file from within the Developer tool. When you edit a schema file, the Developer tool opens the file in the editor you use for .xsd files. You can open the file in a different editor or set a default editor for .xsd files in the Developer tool.

You can use a schema to define element types in a web service. When you update a schema that is included in the WSDL of a web service, the Developer tool updates the web service and marks the web service as changed when you open it. When the Developer tool compares the new schema with the old schema, it identifies schema components through the name attributes.

If no name attribute changes, the Developer tool updates the web service with the schema changes. For example, you edit a schema file from within the Developer tool and change the maxOccurs attribute for element "Item" from 10 to unbounded. When you save the file, the Developer tool updates the maxOccurs attribute in any web service that references the Item element.

If a name attribute changes, the Developer tool marks the web service as changed when you open it. For example, you edit a schema outside the Developer tool and change the name of a complex element type from "Order" to "CustOrder." You then synchronize the schema. When you open a web service that references the element, the Developer tool marks the web service name in the editor with an asterisk to indicate that the web service contains changes. The Developer tool adds the CustOrder element type to the web service, but it does not remove the Order element type. Because the Developer tool can no longer determine the type for the Order element, it changes the element type to xs:string.

Schema Synchronization

You can synchronize a schema object when the schema components change. When you synchronize a schema object, the Developer tool reimports the object metadata from the schema files.

Use schema synchronization when you make complex changes to the schema object outside the Developer tool. For example, you might synchronize a schema after you perform the following actions:

- Make changes to multiple schema files.
- Add or remove schema files from the schema.
- Change import or include elements.

The Developer tool validates the schema files before it updates the schema object. If the schema files contain errors, the Developer tool does not import the files.

To synchronize a schema object, right-click the schema object in the **Object Explorer** view, and select **Synchronize**.

Schema File Edits

You can edit a schema file from within the Developer tool to update schema components.

Edit a schema file in the Developer tool to make minor updates to a small number of files. For example, you might make one of the following minor updates to a schema file:

- Change the minOccurs or maxOccurs attributes for an element.
- Add an attribute to a complex type.
- Change a simple object type.

When you edit a schema file, the Developer tool opens a temporary copy of the schema file in an editor. You can edit schema files with the system editor that you use for .xsd files, or you can select another editor. You can also set the Developer tool default editor for .xsd files. Save the temporary schema file after you edit it.

The Developer tool validates the temporary file before it updates the schema object. If the schema file contains errors or contains components that conflict with other schema files in the schema object, the Developer tool does not import the file.

Note: When you edit and save the temporary schema file, the Developer tool does not update the schema file that appears in the **Schema Locations** list. If you synchronize a schema object after you edit a schema file in the Developer tool, the synchronization operation overwrites your edits.

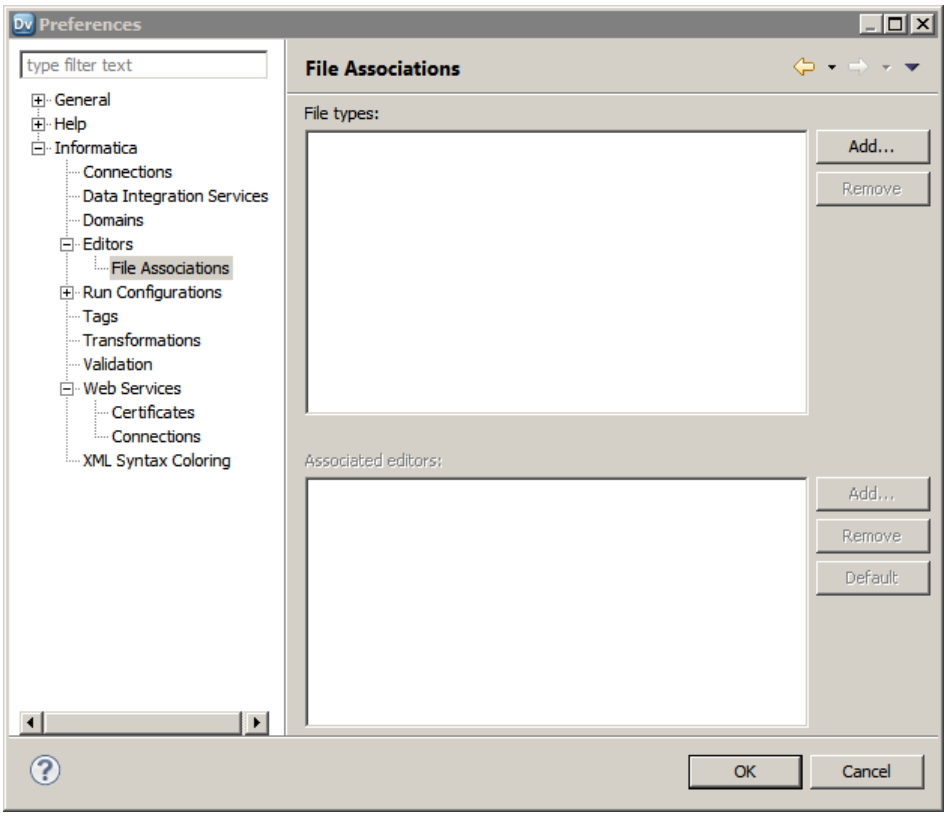
Setting a Default Schema File Editor

You can set the default editor that the Developer tool opens when you edit a schema file.

1. Click **Window** > **Preferences**.

   The **Preferences** dialog box appears.
2. Click **Editors** > **File Associations**.

The **File Associations** page of the **Preferences** dialog box appears.



3. Click **Add** next to the **File types** area.

   The **Add File Type** dialog box appears.

4. Enter `.xsd` as the file type, and click **OK**.
5. Click **Add** next to the **Associated editors** area.

   The **Editor Selection** dialog box appears.

6. Select an editor from the list of editors or click **Browse** to select a different editor, and then click **OK**.

   The editor that you select appears in the **Associated editors** list.

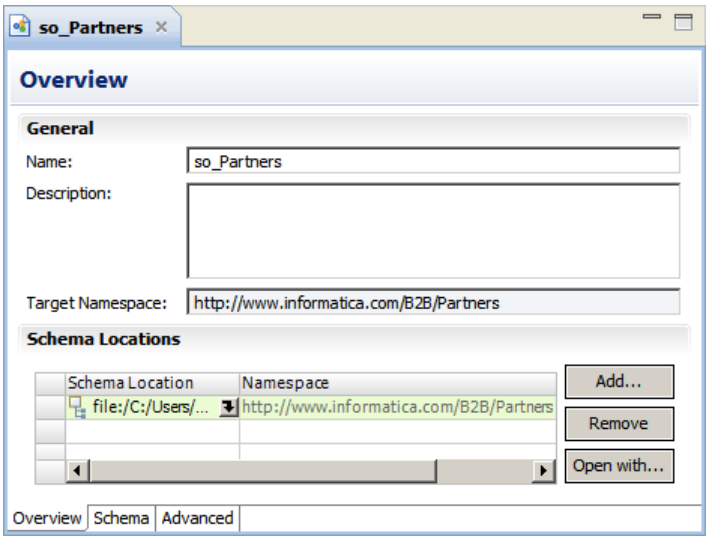
7. Optionally, add other editors to the **Associated editors** list.
8. If you add multiple editors, you can change the default editor. Select an editor, and click **Default**.
9. Click **OK**.

Editing a Schema File

You can edit any schema file in a schema object.

1. Open a schema object.
2. Select the **Overview** view.

The **Overview** view of the schema object appears.



3. Select a schema file in the **Schema Locations** list.
4. Click **Open with**, and select one of the following options:

Option	Description
System Editor	The schema file opens in the editor that your operating system uses for .xsd files.
Default Editor	The schema file opens in the editor that you set as the default editor in the Developer tool. This option appears if you set a default editor.
Other	You select the editor in which to open the schema file.

The Developer tool opens a temporary copy of the schema file.

5. Update the temporary schema file, save the changes, and close the editor.

   The Developer tool prompts you to update the schema object.

6. To update the schema object, click **Update Schema Object**.

   The Developer tool updates the schema file with the changes you made.

CHAPTER 9

Command Line Interface

This chapter includes the following topics:

- Command Line Interface Overview, 137
- CM_console, 137

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]

[ -xoOutputPortName=OutputDocument]

[ -e]
```

Note: Do not include a space between an option and its argument.

The following table describes CM_console options and arguments:

Option	Argument	Description
-	ServiceName	Required. Specifies the name of the service.
-f	InputDocument	Optional. Specifies a path and file name on the local file system. By default, the service uses the document defined in the **example_source** property of the startup component.
-t	InputDocument	Optional. Specifies a string surrounded by double quotes.
-u	InputDocument	Optional. Specifies a URL.
-a	ServiceParameter=InitialValue	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.
-o	FileName [Path]FileName	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.
-r	curr	Optional. Specifies the directory from which you ran the CM_console command.
-r	res	Optional. Specifies the `results` subdirectory under the directory that holds the service in the filesystem repository.
-r	spec=OutputDirectory	Optional. Specifies a directory on the local file system.
-r	guid	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.
-l	UserName	Required when you use HTTP authentication. Specifies the user name for HTTP authentication. **Note:** This option is a lower-case L.
-p	Password	Required when you use HTTP authentication. Specifies the password for HTTP authentication.
-v	-	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.
-S	-	Required if the startup component of the service is a streamer. You must also use the -f option to define the input file.
-xf	InputPortName=InputDocument	Optional. InputPortName specifies the name of an **AdditionalInputPort** 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.
-xt	InputPortName=InputDocument	Optional. InputPortName specifies the name of an **AdditionalInputPort** defined in the service. InputDocument specifies a string surrounded by double quotes. You can enter multiple input ports, separated by spaces.

Option	Argument	Description
-xu	InputPortName=InputDocument	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.
-xo	OutputPortName=OutputDocument	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.
-e	-	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.

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 10

Scripts

This chapter includes the following topics:

- Scripts Overview, 140
- Script Components, 141
- Script Component Properties, 143
- Script Startup Components, 144
- Example Sources, 145
- IntelliScript Editor, 146
- Validate a Script, 147
- Sample Scripts, 147

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:

Component Type	Description
Action	Takes data from a data holder and performs an operation on it. For example, the **RunParser** action runs a Parser.
Anchor	Identifies a section of the input document.
Document processor	Performs a complex transformation on an input document. For example, the **PdfToTxt_4** document processor converts a PDF document to plain text.
Format	Defines the format of the documents for a Parser to process.
Locator	Isolates a single occurrence of a multiple-occurrence data holder.
Mapper	Reads XML documents and writes XML documents. Can be set as the startup component.
Notification	Writes a message to the standard output or to a log. For example, the **XsdValidationError** notification indicates that the input document is not valid as against the schema that defines it.
Parser	Reads documents in any format and writes documents in any format. Can be set as the startup component.
Script Port	Defines an input or output document.
Serializer	Reads XML documents and writes documents in any format. Can be set as the startup component.
Streamer	Breaks large input files into chunks and passes the chunks to a Parser, Mapper, or Serializer. Can be set as the startup component.

Component Type	Description
Transformer	Transforms an input string to an output string. Can be set as the startup component.
Validator	Determines whether input data conforms to a specific data definition.
Variable	Holds data that the Script receives through a service parameter, or holds data from component in the Script.

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.

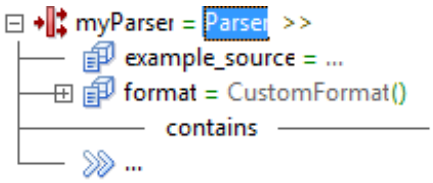
For example, the **example_source** property of a Parser defines sample text that the Parser uses when you run the transformation from the **Data Viewer** view.

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 **Parser** component:



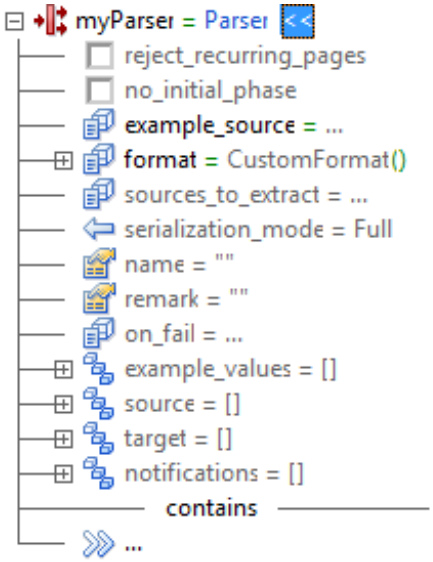
Advanced Properties

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 **Parser** component:



Component Property Values

You set the values for the properties of a component.

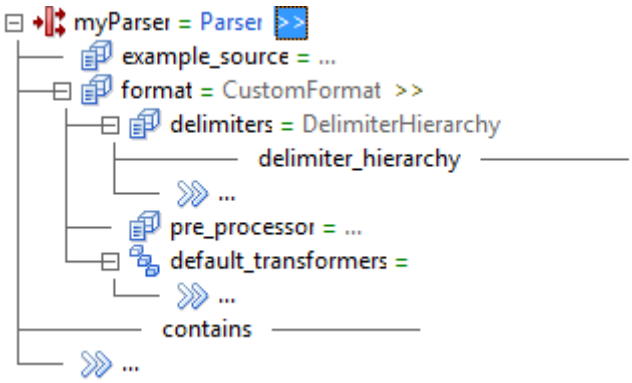
When the value of a property is Boolean, it appears as a check box to the left of the property name. For example, the **optional** property of a **Content** component is Boolean.

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:



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 Example

The following sample text illustrates a part of a local file that you might use for an example source when you parse HL7 documents:

```
MSH|^~\&|ADT1|MCM|FINGER|MCM|198808181126|SECURITY|ADT^A01|MSG00001|P|2.3.1
EVN|A01|198808181123
PID|1||PATID1234^5^M11^ADT1^MR^MCM~123456789^^^USSSA^SS||
SMITH^WILLIAM^A^III||19610615|M||C|1200 N ELM STREET^^JERUSALEM^TN^99999?
1020|GL|(999)999?1212|(999)999?3333||S||PATID12345001^2^M10^ADT1^AN^A|
123456789|987654^NC
NK1|1|SMITH^OREGANO^K|WI^WIFE||||NK^NEXT OF KIN
PV1|1|I|2000^2012^01||||004777^CASTRO^FRANK^J.|||SUR||||ADM|A0
```

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 an input format. The following table describes the input format options:

Option	Description
LocalFile	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.
Text	The **quote** property appears under the **example_source** property. Enter a string.
URL	The **stable_url** property appears under the **example_source** property. Enter a string.

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, 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**.

Validate a Script

When you create a Script, you can validate the Script before you run it. When you validate a Script, the Developer tool checks if there are any failures that could prevent a component from processing data in the expected way.

To validate a Script, in the Developer tool **Outline** view, select the Script, then right-click and select **Validate**. If there are any errors, the **Validation Log** view appears and displays errors or warnings about failures that the validation processor discovered.

If you double-click an error in the **Validation Log** view, the relevant line in the Script is highlighted in the IntelliScript editor.

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:

Script Name	Description
Alternatives	Demonstrates branching and the **Alternatives** anchor.
AppendListItems	Concatenates strings in a multiple-occurrence data holder and demonstrates the **AppendListItems** action.
CalculateValue	Performs a complex numerical computation and demonstrates the **CalculateValue** action.
CombineValues	Concatenates strings and demonstrates the **CombineValues** and **DumpValue** actions.
Content	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.
CopyValue	Copies an entire complex XML element with the **Map** action.
DelimitedSections	Demonstrates the **DelimitedSections** anchor in a Parser.
DocumentOrder	Demonstrates branching and the **Alternatives** anchor, with the **selector** option set to DocumentOrder.
Dynamic_And_RepeatingGroup	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.
EmbeddedParser	Uses an embedded secondary Parser to parse the content of the main Parser and demonstrates the **EmbeddedParser** anchor.

Script Name	Description
EnsureCondition	Evaluates a Boolean JavaScript expression to select alternatives and demonstrates the **EnsureCondition** action.
ManualSerializer	Demonstrates a custom Serializer.
Markers	Demonstrates **Marker** anchors that use the TextSearch, OffsetSearch, TypeSearch, and PatternSearch options.
Marking_Mode	Demonstrates multiple methods of configuring **Marker** anchors.
NonMarker	Demonstrates a Parser that uses only **Content** anchors and searching backward through the input document.
Pattern	Demonstrates extraction of data that matches a restriction defined in the schema.
persistent_search	Demonstrates the **on_partial_match** property of a **Group** and the **adjacent** property of a **Marker**.
ResetListVariable	Resets a list variable using a targetLocator.
RunSerializer	Demonstrates a Parser that calls a secondary Serializer.
HL7	Converts an HL7 file to XML.
TabDelimited	Converts a tab-delimited HL7 file to XML.
Splitter	Splits a file into two files, and demonstrates the **WriteValue** action.
TransformByParser	Uses a Parser to transform specific text to carriage return line feed and demonstrates the **TransformByParser** action.
Transformers_Example	Demonstrates the **Content** anchor with the **value** property set to **LearnByExample**.

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.

CHAPTER 11

Parsers

This chapter includes the following topics:

- Parsers Overview, 149
- Platform-Independent Parsers, 149
- Parser Component Reference, 150

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.

Parser Component Reference

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 313.

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:

Property	Description
example_source	Defines a sample source document to process in the development environment. You can choose one of the following options: - Empty. The Developer tool prompts you for a source document when you run the Parser. - InputPort. Defines an input port. - LocalFile. Defines a file on the local file system. - Text. Defines a string. - URL. Defines a URL. Default is empty. **Note:** If the **sources_to_extract** property is set, the **example_values** property is ignored in the design environment.
example_values	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 **example_values** property only when it processes the example source. It ignores the property when it parses a source document. In the nested **ExampleValue** components, specify the data holders that the calling Parser passes to this Parser and their simulated values.
ExampleValue	Defines an example value under the **example_values** property.
format	Defines the format of the source document. You can choose one of the following options: - BinaryFormat - CustomFormat - HtmlFormat - Rtf Format - TextFormat - XmlFormat Default is CustomFormat. For more information, see "Format Component Reference" on page 177.
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.

Property	Description
notifications	Defines a list of **NotificationHandler** components that the Parser runs on notifications triggered by nested components. For more information, see "Notifications" on page 408.
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 389.
reject_recurring_pages	Determines the number of times the Parser parses the same page. You can choose one of the following options: - Selected. The Parser parses a page only once. - Cleared. The Parser parses a page each time it follows a link to the page. Use **reject_recurring_pages** when a web site contains many links to the same page. **Note:** The **ResetVisitedPages** action resets the history list and allows a Parser to process a page again, even if **reject_recurring_pages** is selected.
remark	A user-defined comment that describes the purpose or action of the component.
serialization_mode	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 325. You can choose one of the following options: - Full. Causes the **Create Serializer** command to copy the non-XML text to the serializer configuration. - 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.
source	Defines a sequence of data holders for input to the Parser. 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. In a secondary Parser, set **Parser** > **source** > **Locator** > **data_holder** to the data holder defined in the associated **AdditionalInputPort** > **data_holder**. For more information, see "Source Property" on page 358.
sources_to_extract	Defines a hard-coded list of source documents that the Parser processes. You can choose one of the following options: - DocList. Defines a list of **LocalFile**, **Text**, and **URL** components. - Empty. The Parser processes the **example_source**. - FileSearch. Defines a folder on the local file system and a file name filter. - InputPort. Defines an input port. Do not use this option. - LocalFile. Defines a file on the local file system. - Text. Defines a string. - URL. Defines a URL. Default is empty. **Note:** Use the **sources_to_extract** property only in the design environment.

Property	Description
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 361.
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.

CHAPTER 12

Script Ports

This chapter includes the following topics:

- Script Ports Overview, 153
- Script Port Component Reference, 153

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:

Property	Description
code_page	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.
data_holder	Defines a data holder where the port stores the content of the input document. Use the same data holder in **Parser** > **source** > **Locator** > **data_holder** property of the associated secondary Parser, Mapper, or Serializer.
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 162. **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:

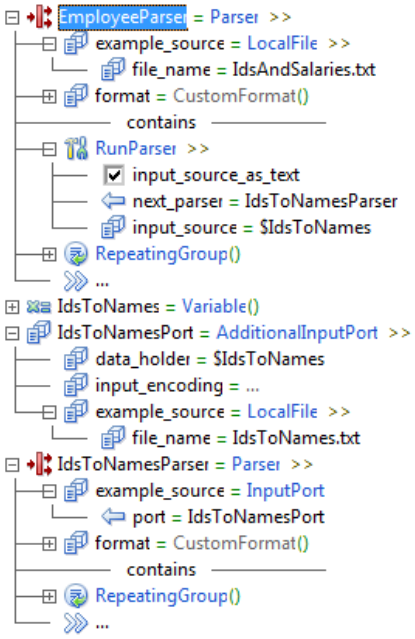
- `IdsAndSalaries.txt` is a table of employee IDs and salaries.
- `IdsAndNames.txt` is a table of employee IDs and names.

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 figure shows an example of a Script with a secondary Parser that references an `AdditionalInputPort` to retrieve the `IdsAndNames.txt` file:



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:

Property	Description
add_BOM_prefix	Adds a byte-order mark (BOM) prefix to the output. The type of BOM prefix is determined by the output encoding defined in the **output_encoding** property. Default is cleared.
code_page	Defines the **encoding** 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_page** property is a child of the **output_encoding** property when it is set to **PortEncoding**. 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.

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.

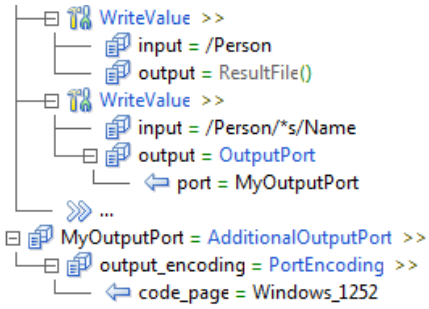
When you deploy the transformation as a service, an application that runs the service can pass the additional output location as a parameter. For example, the location might be a buffer.

Example of AdditionalOutputPort

A Parser generates the following XML structure:

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

The following figure shows a Parser that uses two `WriteValue` actions to generate output.



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

```
<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.

```
<Name>
  <First>Ron</First>
  <Last>Lehrer</Last>
</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:

Property	Description
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 162.

FileSearch

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

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

Property	Description
directory	Defines a folder that contains the input files.
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 162.
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 ***.***.

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:

Property	Description
input	Defines the name of the **AdditionalInputPort** component that defines the input.

LocalFile

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

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

Property	Description
file_name	Defines the path and filename of a file on the local network.
pre_processor	Defines the name of a document processor to apply to the file. For more information, see "Document Processor Component Reference" on page 162.
simulated_url	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.

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:

Property	Description
port	Determines the name of the **AdditionalOutputPort**.

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:

Property	Description
pre_processor	Defines the name of a document processor to apply to the string. For more information, see "Document Processor Component Reference" on page 162.
quote	Defines a text string.

Property	Description
simulated_url	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.
size	Defines a static size for the text buffer. Use the **size** property with binary sources. Default is -1, which means that the buffer is dynamically sized.

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:

Property	Description
post_data	Defines data that the transformation posts to the URL.
pre_processor	Defines the name of a document processor to apply to the files.
retries	Defines the number of retries that the Parser performs before it reports a failure. Default is 0.
seconds_to_wait	Defines the number of seconds to wait between retries. Default is 60.
stable_url	Defines a URL address that contains an input document.

Note: This component is provided for compatibility with projects created in earlier Data Transformation versions. It is being phased out of the Data Transformation system. Do not use it when you develop transformations. Note that a Data Transformation fails to process an HTTPS URL in a Linux environment.

CHAPTER 13

Document Processors

This chapter includes the following topics:

- Document Processors Overview, 161
- Defining a Document Processor, 161
- Document Processor Component Reference, 162
- TextML XML Schema, 171
- PdfToTxt_4 Table Configuration Editor, 172

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 176.

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:

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

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

ExcelToDataXml

The **ExcelToDataXml** document processor converts Microsoft Excel documents to XML.

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

Property	Description
enabled	Determines the content of the output. The **enabled** property has the following options: - Selected. The output contains raw data and formatted data. - Cleared. The output contains only formatted data. Default is selected.
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:

Property	Description
enabled	Defines the value of **param2** or **param3**.
param1	Defines the sheets of the Excel workbook to include in the XML. In the XML output, each sheet is represented by a <sheet> element. **param1** is named **include_sheets** and has the property **value**.
param2	Determines whether the document processor includes empty cells in the output XML if the cells are formatted or merged. **param2** is named **include_empty_cells** and has the 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 the property **enabled**, which has the following options: - Selected. The document processor includes macro code. - Cleared. The document processor omits macro code. Default is cleared.
param4	Determines whether the document processor includes unformatted empty cells in the output XML for cells. **param4** is named **include_empty_non_formatted_cells** and has the property **enabled**, which has the following options: - Selected. The output includes empty cells. - Cleared. The output omits empty cells. 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 97-2003. The processor accesses 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, 2010, or 2013
- XLS files created with Microsoft Excel 2003, 2007, 2010, or 2013

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:

Property	Description
jclass	Defines the path of the Java class.
jmethod	Defines the method to run.

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 422.

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:

Property	Description
import	Defines a DLL in the `ExternLib\Users` folder.

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 424.

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:

Property	Description
param1	The **param1** property is named **validation_params** and has only one property, **value**, which has the following options: - LDNSB - Validator
param2	Defines the validation type. The **param2** property is named **types_to_validate** and has only one property, **value**. Valid values are 1 to 7.
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:

Property	Description
enabled	Defines the value of **param2** or **param4**.
param1	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.
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.

Property	Description
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.
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 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:

Property	Description
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 172.

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 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 171.

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 171.

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.

Note: To use BIRT version 4.5, use the **XmlToDocument_45** preprocessor instead of the **XmlToDocument_372** preprocessor.

The following table describes the properties of the **XmlToDocument_372** document processor:

Property	Description
param1	The path and file name of the BIRT `*.rptdesign` file. The **param1** property is named **report_file** and contains the property **value**, which contains the path and file name.
param2	The format of the output document. The **param2** property is named **output_format** and contains the property **value**, which has the following options: - pdf. PDF document. - doc. Microsoft Word document. - xls. Microsoft Excel workbook. - ppt. Microsoft PowerPoint presentation. - html. HTML web page. - ps. PostScript document. Default is pdf.
param3	A variable that holds the location of the `*.rptdesign` file. The **param3** property is named **report_location** and contains the property **value**, which points to the variable. Default is `$VarServiceInfo/*s/ServiceLocation`.
value	Contains the value of **param1**, **param2**, or **param3**.

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.

XmlToDocument_45

The **XmlToDocument_45** 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) version 4.5 Eclipse add-on to generate the output documents. In BIRT, you configure a report that converts the XML to the desired document format. The **XmlToDocument_45** processor runs the report. For more information about BIRT, see http://www.eclipse.org/birt.

The following table describes the properties of the **XmlToDocument_45** document processor:

Property	Description
param1	The path and file name of the BIRT `*.rptdesign` file. The **param1** property is named **report_file** and contains the property **value**, which contains the path and file name.
param2	The format of the output document. The **param2** property is named **output_format** and contains the property **value**, which has the following options: - pdf. PDF document. - doc. Microsoft Word document. - xls. Microsoft Excel workbook. - ppt. Microsoft PowerPoint presentation. - html. HTML web page. - ps. PostScript document. Default is pdf.

Property	Description
param3	A variable that holds the location of the `*.rptdesign` file. The **param3** property is named **report_location** and contains the property **value**, which points to the variable. Default is `$VarServiceInfo/*s/ServiceLocation`.
value	Contains the value of **param1**, **param2**, or **param3**.

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).

XmlToXlsx

The **XmlToXlsx** document processor converts XML documents to Microsoft Excel .xlsx format. The **XmlToXlsx** document processor can optionally use an .xlsx template to generate the .xlsx document.

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_03_07_10.xsd` schema. You can find the schema in the `doc` subdirectory of the installation directory. The schema file is provided for your information.

The processor reverses the operation of **ExcelToXml_03_07_10**. Use the **ExcelToXml_03_07_10** processor in a Data Processor transformation to transform an Excel workbook to XML. After you process the XML data, use the **XmlToXlsx** processor to transform the data back to an Excel workbook.

This component supports Excel version 2007 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).

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

Property	Description
param1	A variable that holds the name of the `*.xlsx` template file. The **param1** property has the property **template_file** that specifies the template file name.
param2	A variable that holds the location of the `*.xlsx` template file. The **param2** property has the property **template_location** that specifies the template file path.

Note: Excel sheets that exist solely in the template are represented as in the template in the .xlsx output. Excel sheets that are defined in both the template and the XML receive cell styles from the template and cell values from the XML.

To apply a cell style from a different cell in the template, you can use the attribute `style_as` in the XML as part of the `cell` element.

Example

To use the `style_as` attribute to apply a cell style from a cell in the current sheet, set the `style_as` attribute equal to the cell number that contains the style. In the following example, the style from field `A1` in the current sheet applies to the cell defined by the XML context, namely row 3 and cell 2.

```
<row rownumber="3" firstcol="1" lastcol="12" height="405" zeroheight="false" >
            <cell number="2" type="string" style_index="3" font_index="3" style_as="A1">
                <data>Informatica</data>
            </cell>
</row>
```

To apply a cell style from a cell in a different sheet, set the `style_as` attribute equal to the sheet and cell number that contains the style. In the following example, the style is applied from the field `A1` in the workbook sheet named `Summary`.

```
<row rownumber="3" firstcol="1" lastcol="12" height="405" zeroheight="false" >
            <cell number="2" type="string" style_index="3" font_index="3"
style_as="Summary:A1">
                <data>Informatica</data>
            </cell>
</row>
```

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 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>
    <version>2</version>
  </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.

Control or Field	Description
Zoom In	Make the PDF display larger.
Zoom Out	Make the PDF display smaller.
Fit Width	Display the PDF document according to the width of the window.
Prev Page	Go to the previous page.
Next Page	Go to the next page.
Find	Search for a string in the PDF.
Add Table	Add a table to the configuration.
Rem. Table	Remove a table from the configuration.
Add Column	Add a column border to the current table.
Rem. Column	Delete the currently selected column border.
Process	Apply the current table definitions. Click **Process** after every table and column-related action to apply that action.
Tables	A list of tables defined in the input PDF. You can select a table by clicking it.
Name	Name of the currently selected table.
Table Start	An expression defining the upper left corner of the table.
Table End	An expression defining the first text after the table.
Page Header	An expression defining the end of the page header. Use this option to exclude the header from the table processing.
Page Footer	An expression defining the end of the page footer. Use this option to exclude the footer from the table processing.
Use Regular Expressions	If selected, the processor interprets the **Table Start**, **Table End**, **Page Header**, and **Page Footer** as regular expressions and searches for matching text. If not selected the processor interprets these fields as literal text.
Recalculate at Runtime	If you select this option, **PdfToTxt_4** ignores the table configurations that you specified using the table configuration editor. This feature is useful if the tables in a PDF are simple enough for the **PdfToTxt_4** to process without special configuration. For example, suppose a simple PDF financial statement contains a table whose columns may vary slightly from month to month. Select the **Recalculate at Runtime** option to have **PdfToTxt_4** adjust the column widths at runtime.
Recalculate Now	If you have changed the table definition, for example by changing column borders or adding a **Page Header** or **Page Footer**, click **Recalculate Now** to update the table definition.

Control or Field	Description
Page	Number of the PDF page that is currently displayed.
Output as XML	Generates the **PdfToTxt_4** output as XML instead of text.
Delimiter	Enter a character to use as the column separator in the text output. The default is a vertical bar (|).
OK	Click to save the table configuration and return to the Developer tool.
Cancel	Click to return to the Developer tool without saving the table configuration.
Table Navigation Aid	The table navigation aid displays the number of times a table is found in the PDF document. An example of a navigation aid is `Table 'Table 1' found 2 times`. The arrows next to this information let you jump back and forth among the instances of the same table structure.

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**.

CHAPTER 14

Formats

This chapter includes the following topics:

- Formats Overview, 176
- Standard Format Properties, 177
- Format Component Reference, 177
- Delimiters Component Reference, 183
- Format Preprocessor Component Reference, 188

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.

The following table describes the sub-components that you can nest in a format:

Subcomponent	Description
Delimiter	Defines a hierarchy of characters or strings that organize the information in the document, such as newlines and tabs.
Format preprocessor	Cleans up the source before the Parser starts searching for anchors.
Default transformer	Performs predefined operations on the output of each anchor.

Standard Format Properties

The following table describes standard properties of the format components:

Property	Description
default_Transformers	Defines a list of Transformers that the Parser applies to the output of each content anchor.
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 183.
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:

Property	Description
default_transformers	Defines a list of Transformers that the Parser applies to the output of each content anchor. Default is empty.
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 183. Default is Positional.
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.

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:

Property	Description
default_transformers	Defines a list of Transformers that the Parser applies to the output of each content anchor. Default is empty.
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 183. 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:

```
Ron      Lehrer && 547329876:27
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:

```
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:

Property	Description
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 183. 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. - RtfProcessor. Normalizes RTF files. Default is HtmlProcessor.
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:

Property	Description
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: - RtfToASCII. Removes RTF control words from the output. - RemoveRtfFormatting. Removes RTF formatting instructions from the text. - 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 183. Default is RTF.
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 RtfProcessor.
remark	A user-defined comment that describes the purpose or action of the component.

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:

Property	Description
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: - 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 183. 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.

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:

Property	Description
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 XML tags from the output. - HtmlEntitiesToASCII. Converts XML 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 183. 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. - RtfProcessor. Normalizes RTF files. Default is HtmlProcessor.
remark	A user-defined comment that describes the purpose or action of the component.

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>jkl<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

The **Delimiter** 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:

Property	Description
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 240.

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:

Property	Description
opening	Defines the opening delimiter.
closing	Defines the closing delimiter.
escape_sequence	Defines a prefix that causes the Parser to ignore an instance of the opening or closing delimiter in the source document.

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 201.

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
     xyz
ghi      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:

```
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:

```
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 161.

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 15

Data Holders

This chapter includes the following topics:

- Data Holders Overview, 189
- XML Schemas, 189
- Using a Schema to Map Anchors, 192
- Generating Valid XML, 193
- Variables, 194
- Variable Component Reference, 198
- Multiple-Occurrence Data Holders, 199

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, XMap, 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.

Add the schema to the Model repository. You can then map the content of a document to elements and attributes that are defined in the schema.

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 192.

Unsupported Schema Features

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

Feature	Limitation
Uniqueness constraints	The **unique**, **key**, and **keyref** elements are ignored. The event log includes a warning.
Default values for elements of mixed type	The Script ignores the default. The event log includes a warning.
Default data type	If the type of an element is undefined, the Script processes it as `xs:string`. The event log includes a warning. You can change the default to `xs:anyType`.

Feature	Limitation
Regular expressions	There are minor discrepancies between the regular expression processor and the schema standard.
Sequence defining multiple elements having the same name	If an `xs:sequence` contains multiple `xs:element` definitions having the same name, the Script processes only the first `xs:element`. The event log includes a warning. To resolve the problem, wrap each `xs:element` in an independent `xs:sequence`.
Minimum and maximum dates	If a facet defines a minimum or maximum value for an `xs:date` element, the transformation fails.
Lax or skip validation options	In an `xs:any` or `xs:anyAttribute` element, the Script ignores a **processContents** value of **lax** or **skip**. It behaves as if the value were **strict**.
Substitution group	The Script permits a **substitutionGroup**, even if a **block** or **blockDefault** attribute forbids substitutions.
XSI type	The Script permits an `xsi:type` attribute even if a **block** attribute of forbids it.
Built-in types	Some built-in types do not have correct patterns, for example, when they include characters above ASCII 127.
Substitution group without a type	The Script sometimes fails when a substitution group does not have a type.
Empty namespace	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 **Locator** and the **Locator** fails.
List	The Script reads a space-separated `xs:list` as a single item, which might fail if its length exceeds the stated limit for individual items in the list.
Floats and doubles	`xs:float` and `xs:double` do not accept valid values of **INF**, **-INF**, or **NaN**.
Element with both fixed and mixed attributes	The Script does not read all parts of an element that has both fixed and mixed attributes.
max_occurs=0	The Script creates output even when `max_occurs=0`.
Token	The Script does not parse `xs:token` that contains tabs, carriage returns, or line feeds.
Normalized string	The Script does not load an XML when an `xs:normalizedString` contains tabs, carriage returns, or line feeds.

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 map `Content` anchors to output data holders. When you define a Serializer, you 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:

```
data_holder = /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:

```
<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:

```
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:

```
<!-- 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:

```
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 213.

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:

```
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.

The discussion presents examples of the behavior.

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 199.

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 211.

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 **XmlToDocument** processor:

Variable	Description
VarServiceInfo > *ServiceLocation*	The directory path of the Script or service that is currently running.

Read-Only Access Variables

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

Variable	Description
VarRequestedURL	The path of the source document that a Parser is processing.
VarCurrentURL	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.
VarCurrentPost	The form data that a Parser submitted to retrieve the current page.

Read-Only System Time Variables

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

Variable	Description
VarSystem > **ExecStartTime** > *Year*	Year when the transformation began execution
VarSystem > **ExecStartTime** > *Month*	Numerical month

Variable	Description
VarSystem > **ExecStartTime** > *MonthName*	Name of month
VarSystem > **ExecStartTime** > *Day*	Day of month
VarSystem > **ExecStartTime** > *DayName*	Day of week
VarSystem > **ExecStartTime** > *Hour*	Hour
VarSystem > **ExecStartTime** > *Minute*	Minute
VarSystem > **ExecStartTime** > *Second*	Second
VarSystem > **ExecStartTime** > *Millisecond*	Millisecond

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 389.

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:

Variable	Description
VarLastFailure > *InternalId*	Failure identifier
VarLastFailure > *Text*	Failure description
VarLastFailure > *Location*	Location of the failure in the Script
VarLastFailure > *AnchorName*	Name of the component that failed
VarLastFailure > *Data*	Additional information about the failure
VarServiceInfo > *ServiceName*	Name of the service
VarServiceInfo > **StandardError** > *StandardErrorDir*	Directory path of the user log
VarServiceInfo > **StandardError** > *StandardErrorName*	File name of the user log

Variables Used for Structured Parsing

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

Variable	Description
VarStructureDetails > *Name*	The **name** property of the subelement that matches the record.
VarStructureDetails > *Repetitions*	The iteration number of the record.
VarStructureDetails > *RecordIndex*	The index number of the record in the overall **StructureDefinition** input.
VarStructureDetails > *RecordId*	The recorfd identifier. If there are multiple identifiers, the variable contains a comma-separated list.
VarStructureDetails > *InternalPath*	Internal information, not for use.

For more information, see "StructureDefinition" on page 236.

Variables Used in Notifications

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

Variable	Description
VarNotificationDetails > *Name*	The name of the notification.
VarNotificationDetails > *Path*	The XPath of the data holder to which the notification applies. For example, if a validator triggers a notification in a **Content** anchor, the **Path** is the data holder where the **Content** anchor stores its output.
VarNotificationDetails > *Value*	The input value that caused the notification. If a validator triggers the notification, the **Value** is the invalid input data. If a **Notify** action triggers the notification, you can specify the **Value** in the **Notify** configuration.
VarNotificationDetails > *Creator*	The location in the Script that triggered the notification.

For more information, see "Notifications" on page 408.

Mapping Anchors to Variables

You can map a **Content** anchor to a variable in the same way that you map to any other data holder.

Do not map an anchor to a read-only system variable.

Using Variables in Actions

Variables are often used as the input of actions. You can use a variable in the same way as you use other data holders. For more information, see "Actions Overview" on page 289.

Initializing Variables at Runtime

You can initialize the values of variables in the following ways:

- In the Script, you can configure the **initialization** property of the **Variable**.

  The initial values that you set by this approach are used when you run the transformation in the Developer tool or as a service.

- An application can pass the initial values as service parameters to a service at runtime.

  The service parameters override the **initialization** property of the variables. If the Script specifies an initial value, and you also pass a value from an application, the latter value is used.

Variable Component Reference

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

For more information about system variables, see "System Variables" on page 195.

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 198.

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

Property	Description
initialization	Defines an initial value for the variable. You can initialize variables that have simple data types. Default is empty.
InitialValue	Defines an initial value for the variable. **InitialValue** has one property, **value**.
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 199.
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:

```
Jack Jennie Larissa
```

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:

```
<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:

```
<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 352.

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:

```
<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 211.

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 352.

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:

```
<xs:schema xmlns:xs="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 352.

   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 16

Anchors

This chapter includes the following topics:

- Anchors Overview, 201
- Mapping Content Anchors to Data Holders, 202
- Defining Anchors, 203
- Standard Anchor Properties, 205
- How a Parser Searches for Anchors, 206
- Anchor Component Reference, 212
- Searcher Component Reference, 240
- Anchor Subcomponent Reference, 244

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 176.

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 206.

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 211.

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:

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

For more information, see "Multiple-Occurrence Data Holders" on page 199.

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 192.

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:

Anchor	Text in the Source Document
`Marker`	`First Name`
`Content`	`Ron`
`Marker`	`Last Name`
`Content`	`Lehrer`

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 206.

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.

Standard Anchor Properties

The following table describes standard properties of anchors:

Property	Description
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 206.
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 206.
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 408.
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 389.

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 389.
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 206.
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:

Anchor	Text in the Source Document	Phase
Marker	First name:	Initial
Content	Ron	Main
Marker	Last name:	Initial
Content	Lehrer	Main

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.

Anchor	Text in the Source Document	Phase	Search Scope	Search Criteria
`Marker`	`First name:`	`Initial`	Entire document	Text = `First name:`
`Content`	`Ron`	`Main`	End of `First name:` to start of `Last name:`	After the leading space Before the next space
`Marker`	`Last name:`	`Initial`	End of `First name:` to end of document	Text = `Last name:`
`Content`	`Lehrer`	`Main`	End of `Last name:` to end of document	After the leading space Before the next space

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.

Marking Property	Explanation	Result
full	The Script places reference marks at the beginning and end of the current anchor. This is the default behavior.	The Script seeks the next anchor after the end of the current anchor. `Content B` follows `Content A`.
begin position	The Script places a reference mark only at the start of the current anchor.	The Script seeks the next anchor after the start of the current anchor. `Content B` overlaps or follows `Content A`.
end position	The Script places a reference mark only at the end of the current anchor.	The Script seeks the next anchor after the end of the current anchor. `Content B` follows `Content A`.
none	The Script does not place any reference marks at the current anchor.	The Script seeks the next anchor after the end of the preceding anchor. `Content B` follows `Marker`, without regard to `ContentA`.

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 240.

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

```
, 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 teh 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 252.

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:

Property	Description
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 206.
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 389.
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 389.
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 206. Default is main.

Property	Description
remark	A user-defined comment that describes the purpose or action of the component.
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="PrincessWeds.html">Norwegian Princess Weds</a> - News
<a href="BanksMerge.html">Local Banks to Merge</a> - Business
<a href="HomeTeamWins.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.cmw`. 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:

Property	Description
allow_empty_values	Determines whether the **Content** anchor can be empty. You can choose one of the following options: - Selected. The **data_holder** is assigned an empty value. - Cleared. Empty values are not allowed. **allow_empty_values** must be selected in the following situations: - When the anchor is configured with **value = LearnByExample** and there is nothing between the delimiters. - When there is nothing between the **opening_marker** and the **closing_marker**.
closing_marker	Defines the end of a region where the Parser searches for the **Content** anchor. You can choose one of the following options: - NewlineSearch. The end of the **Content** anchor is the next newline character. - OffsetSearch. The end of the **Content** anchor is the number of characters specified in **offset**. - PatternSearch. The end of the **Content** anchor is the first text that matches a specified regular expression. - TextSearch. The end of the **Content** anchor is a specified text string.
data_holder	Defines a data holder where the **Content** anchor stores the retrieved text.
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 206.

Property	Description
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 211.
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 252.
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 206.
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 389.
opening_marker	Defines the start of a region where the Parser searches for the **Content** anchor. The possible values are the following components: - NewlineSearch. The start of the **Content** anchor is the next newline character. - OffsetSearch. The start of the **Content** anchor is the number of characters specified in **offset**. - PatternSearch. The start of the **Content** anchor is the first text that matches a specified regular expression. - TextSearch. The start of the **Content** anchor is a specified text string.
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 389.

Property	Description
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.
transformers	Defines a sequence of transformers that the Parser applies to the retrieved text. For more information, Chapter 17, "Transformers" on page 252.
validators	Defines a list of validators applied to the data. For more information, see "Validators" on page 392.
value	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 206. You can choose one of the following options: - Empty. The **Content** anchor retrieves the entire search scope. - 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. - 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. - PatternSearch. The **Content** anchor retrieves the first text that matches a specified regular expression. - TypeSearch. The **Content** anchor retrieves the first text that matches a specified data type. Default is empty. For more information about these options, see the "Searcher Component Reference" on page 240. 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 211.

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 240.

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.cmw`. 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:

Property	Description
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 206.
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 408.

Property	Description
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 389.
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 389.
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 206. 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|`
	- around. There are separators before and after each section, including the first and last sections. For example,
	`|1|2|3|4|`
	- before. There is a separator before each section, including the first section. For example,
	`|1|2|3|4`
	- between. There is a separator between the successive sections, but not before the first section and not after 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.cmw`. 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:

Property	Description
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 206.
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 389.
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 389.
Parser	Determines the name of a secondary Parser that is defined in the same project.
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 206. Default is main.

Property	Description
remark	A user-defined comment that describes the purpose or action of the component.
schema_connections	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 247.
source_transformers	Defines a sequence of transformers that the Parser applies to the search scope before the secondary Parser processes it.

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.cmw`. 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:

Property	Description
closing	Defines the closing anchor of the **EnclosedGroup**.
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 206.
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
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 408.
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 389.
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 389.
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 206. 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 358.
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 361.

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:

Property	Description
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 389.

Property	Description
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 206. 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:

Property	Description
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 206.
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 389.

Property	Description
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 389.
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.
replace_with	Defines a literal replacement string or a data holder that contains the replacement string.
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 358.
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 361.

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 284.

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.

Group

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:

Property	Description
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 206.
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
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 408.
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 389.
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.
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 389.
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 206. 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.

Property	Description
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. - LocatorByOccurence. 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 358.
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. - LocatorByOccurence. 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 361.

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 206.

The following table describes the properties of the **Marker** anchor:

Property	Description
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 206.
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
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 206.
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 389.
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 389.
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 206. 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 206. 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 240.

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 203.

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:

Property	Description
count	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.
current_iteration	Defines a data holder where the **RepeatingGroup** outputs the number of the current iteration.
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.
iteration_order	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.
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 206.
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
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 408.
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 389.
on_iteration_fail	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 **on_fail** property to write an entry if the entire **RepeatingGroup** fails. For more information, see "Failure Handling" on page 389.
on_partial_match	Defines the behavior when some, but not all, of the required anchors nested under the **RepeatingGroup** appear in the input. The **on_partial_match** property has the following options: - fail. The iteration fails. - skip. **RepeatingGroup** 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.
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 389.
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.

Property	Description
search_order	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.
separator	Defines an anchor that delimits the sections. If you leave the **separator** property empty, the **RepeatingGroup** 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 **Group** in the **separator** property instead of a **Marker**.

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|`
	- around. There are separators before and after each section, including the first and last sections. For example,
	`|1|2|3|4|`
	- before. There is a separator before each section, including the first section. For example,
	`|1|2|3|4`
	- between. There is a separator between the successive sections, but not before the first section and not after the last section. For example,
	`1|2|3|4`

skip_failed_iterations	Determines whether failed iterations are skipped. You can choose one of the following options: - Selected. **RepeatingGroup** skips over a failed iteration and proceeds with the next iteration. If an iteration succeeds, the **RepeatingGroup** succeeds. - Cleared. **RepeatingGroup** fails if any iteration fails. The **skip_failed_iterations** property has an effect only if **separator** is defined. Default is selected.
source	Defines a sequence of data holders for input to the **RepeatingGroup**. 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 **RepeatingGroup** is called by another component. For more information, see "Source Property" on page 358.
target	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: - 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 **RepeatingGroup** is used by another component. For more information, see "Target Property" on page 361.

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 206.

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.
- If the `RepeatingGroup` is not configured with a `separator`, it searches only for the anchors.

End of a RepeatingGroup

You can signal the end of a `RepeatingGroup` in ways such as the following:

- The `RepeatingGroup` can continue until the end of the document.
- You can insert a `Marker` after the `RepeatingGroup`. By default, the `Marker` is in an earlier search phase than the `RepeatingGroup`. This causes the Parser to search for the `Marker` first and use it to limit the search scope of the `RepeatingGroup`. For more information, see "Adjusting the Search Phase" on page 208.
- You can set the `count` property, limiting the search to a maximum number of iterations.
- 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 has delimited records. You organize the input records in predefined ways. For example, a record of type A precedes a record of type B, followed 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:

Property	Description
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_definition	Defines a list of anchors and actions that identify and extract the records. The list must contain an **ExtractRecord** anchor.
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 408.
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 206. Default is main.
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.

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.

Note: When a Library transformation has a **StructureDefinition** with a **format_definition** that contains a **RepeatingGroup**, the transformation overwrites the repeating element instead of iterating the element.

The subelements hierarchy reflects the required organization of the records. You can configure sequences, choices, loops of records, and required or 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. The transformation uses the **StructureDefinition** anchor to find and diagnose input errors. Usually the **StructureDefinition** anchor can continue to parse the remainder of the input.

Example

A Parser processes input with the following structure:

```
ST*12*23~
NM1*12*23*4~
N1*12*23~
NM1*13*23*4~
N2*1*2*3~
SE*12*2:3:4~
```

The records are delimited by newline characters. Within each record, the fields are delimited by `~`, `*`, and `:` characters.

The first field of each record identifies the record type. There are five main record types:

```
ST
NM1
N1
N2
SE
```

In the `NM1` records, the second field is a subtype. There are two subtypes:

```
NM1*12
NM1*13
```

The records must occur in the following sequence:

1. An `ST` record.
2. An `NM1*12` record followed by `N1`.
3. An `NM1*13` record followed by `N2`.
4. An `SE` record.

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:

Subelement	Description
RecordStructureLocal	Matches and parses a single record.
SequenceStructureLocal	Defines a sequence of nested subelements. The records must occur in the same sequence as the nested subelements.
ChoiceStructureLocal	Defines a choice of nested subelements. A record must match one of the nested subelements.
AllStructureLocal	Defines a set of nested subelements, without a specified sequence. The records can match the nested subelements in any order.

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 244.

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:

Notification	Description
MandatoryStructureMissing	A mandatory record does not appear in the input.
MismatchIDs	The record and subelement IDs partially match. For example, there are two record identifiers, and only one of them matches.
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. For example, the subelements define a sequence `ABC`, but the input contains `ACB`.
UnexpectedRecord	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`.
UnrecognizedRecord	No subelement matches any of the record identifiers.
XsdValidationError	The input does not match the requirements of the schema.

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 195.

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 212.
- To define delimiter characters or strings. For more information, see "Format Component Reference" on page 177.
- To define the `find_what` string of a **Replace** transformer. For more information, see "Transformer Component Reference" on page 254.

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 215.

The following table describes the properties of the **AttributeSearch** searcher component:

Property	Description
att	Defines the name of the attribute.
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:

```
<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`.

```
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 215.

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:

Property	Description
example	Defines the text in the example source document at the anchor location.

Note: The **LearnByExample** searcher component applies heuristics that are sensitive. To use the **LearnByExample** searcher component, the example must have the same form as the expected input. The form must be uniform across the entire input file or files. If the example is not the same as the input, the **LearnByExample** searcher component might fail.

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:

Property	Description
allow_smaller_offset	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. **OffsetSearch** succeeds when an offset extends beyond the search scope. - Cleared. **OffsetSearch** fails when an offset extends beyond the search scope.
offset	Defines the number of characters between the reference point and the anchor. In some locations where **OffsetSearch** is used, such as in a **Marker** anchor, the IntelliScript editor displays a browse button next to the **offset** property. You can enter a value or browse to a data holder containing the value.

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:

Property	Description
escape_sequence	Defines a prefix that causes the search component to ignore an instance of the pattern in the source document.
pattern	Defines the regular expression.

For more information about the syntax of regular expressions, see "Regular Expression Syntax" on page 276.

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:

Property	Description
opening	Defines the search criterion for the opening marker. The options are the following searcher components: - NewlineSearch - OffsetSearch - PatternSearch - TextSearch
closing	Defines the search criterion for the closing marker. The options are the following searcher components: - NewlineSearch - OffsetSearch - PatternSearch - TextSearch

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:

Properties	Description
escape_sequence	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.
match_case	Determines whether the defined **text** must match exactly, with the same uppercase and lowercase letters. Default is cleared.
text	Defines the string to find. In locations where dynamic search is supported, you can browse to a data holder that contains the string.

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 232.

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 215.

The following table describes the properties of the **TypeSearch** searcher component:

Property	Description
val_type	Determines the data type of the anchor to search for.

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 236.

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:

Property	Description
action	Defines an action that runs on the list of sub-components.
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 408.

Property	Description
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 236.

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:

Property	Description
action	Defines an action that runs on the list of sub-components.
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 408.
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.

ChoiceStructure

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 236.

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:

Property	Description
action	Defines an action that runs on the list of sub-components.
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 408.
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 236.

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:

Property	Description
action	Defines an action that runs on the list of sub-components.
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 408.
remark	A user-defined comment that describes the purpose or action of the component.

Property	Description
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:

Property	Description
data_holder	Defines a data holder that is referenced in the main Parser, Serializer, or Mapper. **Note:** 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 `xs:boolean`, ensure that the variable has a value before you run the embedded transformation.
embedded_data_holder	Defines a data holder that is referenced in the secondary Parser, Serializer, or 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.
remark	A user-defined comment that describes the purpose or action of the component.

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 336.

EmbeddedStructure

The **EmbeddedStructure** component activates components defined at the global level of the Script. For more information, see "StructureDefinition" on page 236.

EmbeddedStructure is a sub-element of the **StructureDefinition** anchor.

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

Property	Description
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 408.
ref	Defines the name of the globally configured component.
remark	A user-defined comment that describes the purpose or action of the component.
schema_connections	Connects the target of the references subelement with the target of the **EmbeddedStructure**. For more information, see "Connect" on page 247.
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 236.

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:

Property	Description
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 one or more strings.
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 408.

Property	Description
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.

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 236.

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:

Property	Description
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 one or more strings.
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 408.
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.

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 236.

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:

Property	Description
action	Defines an action that runs on the list of sub-components.
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 408.
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.

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 236.

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:

Property	Description
action	Defines an action that runs on the list of sub-components.
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 408.
remark	A user-defined comment that describes the purpose or action of the component.

Property	Description
sub_elements	Defines a list of nested sub-elements.
target	Defines a data holder where the component stores its output.

CHAPTER 17

Transformers

This chapter includes the following topics:

- Transformers Overview, 252
- Defining Transformers, 252
- Standard Transformer Properties, 254
- Transformer Component Reference, 254

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 177.

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:

```
<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 289.

Standard Transformer Properties

The following table describes standard properties of Transformers:

Property	Definition
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.
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 389.

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:

Property	Description
base_URL	Defines the base path or URL.
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 389.
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:

Property	Description
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:

Property	Description
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 389.
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:

Property	Description
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	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 389.
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_white_spaces. Processes all characters except whitespace. Default. - 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:

Property	Description
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 389.
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:

Property	Description
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

```
<![CDATA[100 < 200]]>
```

to

```
100 < 200
```

Note: If you write the result to XML, the Script re-encodes it using the standard XML encoding:

```
100 &lt; 200
```

The following table describes the properties of the **CDATADecode** transformer:

Property	Description
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 389.
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
```

to

```
<![CDATA[100 < 200]]>
```

The following table describes the properties of the **CDATAEncode** transformer:

Property	Description
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 389.
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:

Property	Description
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 260.

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:

Property	Description
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 389.
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:

Property	Description
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
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 389.
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:

```
input_format = "d/M/yy"
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

Pattern Symbol	Meaning	Type	Examples
`G`	Era designator	Text	`AD`
`y`	Year	Number	`1996`
`u`	Extended year	Number	`-200`, meaning 201 BC
`M`	Month in year	Text or number	`July 07`
`d`	Day in month	Number	`10`
`h`	Hour in AM/PM (1-12)	Number	`12`
`H`	Hour in day (0-23)	Number	`0`
`m`	Minute in hour	Number	`30`
`s`	Second in minute	Number	`55`

Pattern Symbol	Meaning	Type	Examples
`S`	Fractional second	Number	`978`
`E`	Day of week	Text	`Tuesday`
`e`	Day of week (local 1-7)	Number	`2`
`D`	Day in year	Number	`189`
`F`	Day of week in month	Number	`2`, meaning the 2nd Wednesday in July
`w`	Week in year	Number	`27`
`W`	Week in month	Number	`2`
`a`	AM/PM marker	Text	`PM`
`k`	Hour in day (1-24)	Number	`24`
`K`	Hour in AM/PM (0-11)	Number	`0`
`z`	Time zone	Time	`Pacific Standard Time`
`Z`	Time zone (RFC 822)	Number	`-0800`
`v`	Time zone (generic)	Text	`Pacific Time`
`g`	Julian day	Number	`2451334`
`A`	Milliseconds in day	Number	`69540000`
`' '`	The text within single quotes is interpreted as a literal string	Text	`'Today is 'dd/MM/yyyy` generates output such as `Today is 15/03/2005`
`''`	Literal single quote	Text	`'o''clock'` generates the output `o'clock`

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:

Property	Description
table	Defines the data holder that contains the 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:

Property	Description
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 389.
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.
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.
param3	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.

Property	Description
param4	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.
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 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

```
http://www.example.com?name=John Doe
```

to

```
http://www.example.com?name=John%20Doe
```

Note: Parenthesis characters are not encoded.

The following table describes the properties of the **EncodeAsUrl** transformer:

Property	Description
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:

Property	Description
add_prefix	Adds a Byte Order Mark (BOM) when the output encoding is UTF-8 or UTF-16.
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 389.
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:

Property	Description
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 389.
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 424.

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:

Property	Description
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 389.
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:

Property	Description
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 389.
remark	A user-defined comment that describes the purpose or action of the component.
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:

Property	Description
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 389.

Property	Description
remark	A user-defined comment that describes the purpose or action of the component.
signed	Determines whether the output number has a sign. You can choose one of 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 input. The supported values are 1 to 8. Default is 1.
to_base	Defines the base of the output. You can choose one of the following options: - decimal. Base 10. - hexadecimal. Base 16 using capital letters A-F. - lowercase hexadecimal. Base 16 using lowercase letters a-f. - octal. Base 8. Default is decimal.

Note: This component does not support UTF-8 input encoding.

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:

Property	Description
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 389.
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:

Property	Description
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_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 389.
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.

HtmlEntitiesToASCII

The **HtmlEntitiesToASCII** transformer converts HTML entities to plain text. For example, it converts `©` or `©` to the copyright symbol (©).

The following table describes the properties of the **HtmlEntitiesToASCII** transformer:

Property	Description
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/sgml/entities.html. The supported entities include:

- `&`, `<`, `>`, and `"` (`& < > "`, respectively)
- Numeric character codes `&#0;` to `ÿ`
- Entities for Latin-1 characters: ` ` = non-breaking space, `©` = 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 188.

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:

Property	Description
digits_after_decimal_point	Determines the number of digits after the decimal point.
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 389.
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:

Property	Description
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:

Property	Description
Entry	Defines a **key** and **value** pair.
key	Defines a unique input string.
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:

Property	Description
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 389.
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 422.

LookupTransformer

The **LookupTransformer** transformer looks up a value in a table.

The following table describes the properties of the **LookupTransformer** transformer:

Property	Description
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 263. - 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 271. - 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 287. - [TableName]. A **DynamicTable**, **InlineTable**, or **XMLLookupTable** defined at the global level of the Script. 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. Default is cleared. For more information about component failure, see "Failure Handling" on page 389.
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.

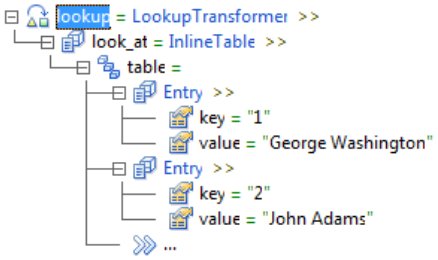
For example, you can configure a **LookupTransformer** to look up values in the following table:

Key	Value
1	George Washington
2	John Adams
3	Thomas Jefferson
4	James Madison

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:



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:

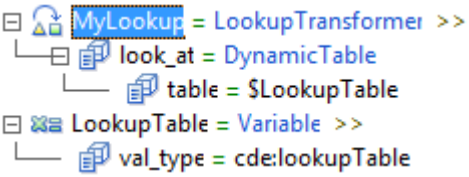
```
<?xml version="1.0" encoding="windows-1252" ?>
<lt:LookupTable xmlns:lt="http://www.Itemfield.com/Engine/V4/lookupTable"
matchCase="false">
  <lt:Entry key="1" value="George Washington" />
  <lt:Entry key="2" value="John Adams" />
</lt:LookupTable>
```

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.



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:

Property	Description
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:

Property	Description
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 389.
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:

```
transformer
```

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:

```
Xformer
```

Regular Expression Syntax

A regular expression defines a search pattern according to a standard syntax.

The Data Processor transformation uses the Regex++ implementation of regular expressions, © 1998-2003 by Dr. John Maddock, Version 1.33, 18 April 2000.

Note: Regex++ does not support locales.

The following table lists some special characters that you can use in regular expressions:

Character	Meaning	Example
*	Matches zero or more instances of the preceding character.	`ab*c` matches `ac`, `abc`, or `abbbc`.
?	Matches zero or one instance of the preceding character.	`ab?c` matches `ac` or `abc`.
+	Matches one or more instances of the preceding character.	`a+` matches `a` or `aaaa`.
{}	Matches the specified number of instances of the preceding character.	`ab{2}c` matches `abbc`.
[]	Matches any of a set of characters.	`a[bst]c` matches `abc`, `asc`, or `atc`.
-	Defines a range of characters inside square brackets.	`[A-Za-z]` matches any character in the English alphabet. `[A-Za-zü]` matches any character in the English alphabet or the German `ü`.
.	Matches any single character.	`a.c` matches `abc`, `a c`, or `a1c`.
^	Matches the start of the input text.	`^P.` matches `Pe` but not `Pi` in "Peter Piper."
$	Matches the end of the input text.	`r.$` matches `rs` in "Peter Piper's peppers."
|	Matches either of two expressions.	`abc|ded` matches `abc` or `def`.
()	Grouping	`A(abc|def)` matches `Aabc` or `Adef`.
\	Escapes one of the other special characters, treating it as a literal character.	`\.` matches a literal period, rather than any character.

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:

```
exp = abc([0-9]+)(def)
replacement = $1
```

This replaces `abc5624def` with `5624`.

Alternatively, suppose you set:

```
exp = abc([0-9]+)(def)
replacement = $2ZYX$1
```

This replaces `abc5624def` with `defZYX5624`.

RemoveMarginSpace

The **RemoveMarginSpace** transformer deletes leading and trailing space characters from the text.

The following table describes the properties of the **RemoveMarginSpace** transformer:

Property	Description
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:

Property	Description
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:

Property	Description
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 389.
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:

Property	Description
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. Default is TextSearch. For more information, see the "Searcher Component Reference" on page 240.
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`.

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 389.
remark	A user-defined comment that describes the purpose or action of the component.
replace_with	Defines the replacement string.

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:

Property	Description
align	Defines the text alignment within the resized string. You can choose one of the following options: - left. Padding or trimming is on the right. - right. Padding or trimming is on the left.
allow_smaller_size	When selected, the Rezise transformer fits the input text to a specified size with padding if the string is smaller than the size defined by the **size** parameter. If not selected, and the input string is smaller than the specified size, the transformer fails.
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.
padding_character	Defines the padding character. Type the character or select a data holder that contains a character.
remark	A user-defined comment that describes the purpose or action of the component.
size	Defines the size of the output text. Type an integer or select a data holder that contains an integer.

ReverseTransformer

The **ReverseTransformer** transformer reverses a string. For example, it transforms `1234` to `4321`.

The following table describes the properties of the **ReverseTransformer** transformer:

Property	Description
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 389.
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 188.

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:

Property	Description
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:

Property	Description
begin	Defines the start location. 0 means to start at the beginning of the input.
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.
end	Defines the end location.
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.

ToFloat

The **ToFloat** 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 **ToFloat** transformer:

Property	Description
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 389.
remark	A user-defined comment that describes the purpose or action of the component.
size	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.

Note: This component does not support UTF-8 input encoding.

ToInteger

The **ToInteger** 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:

Property	Description
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: - decimal. Base 10. - hexadecimal. Base 16 using capital letters A-F. - lowercase hexadecimal. Base 16 using lowercase letters a-f. - octal. Base 8. 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 389.
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:

Property	Description
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 389.
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:

Property	Description
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 260.
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
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 389.
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:

Property	Description
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 389.
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 `<installation>\client\DT\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.

Note: The samples are only available when you perform a client installation.

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 161.

The following table describes the properties of the **TransformByProcessor** transformer:

Property	Description
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 389.
processor	Defines the name of the document processor.
remark	A user-defined comment that describes the purpose or action of the component.

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:

Property	Description
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. Default is cleared. For more information about component failure, see "Failure Handling" on page 389.
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:

Property	Description
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	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 389.
remark	A user-defined comment that describes the purpose or action of the component.

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:

Property	Description
xml_file_name	Defines the path and file name of the XML file.

The following XML document is valid against the schema:

```
<?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:

Property	Description
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	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 389.
remark	A user-defined comment that describes the purpose or action of the component.
xslt_file	Defines the path and file name of the XSLT file.

CHAPTER 18

Actions

This chapter includes the following topics:

- Actions Overview, 289
- Standard Action Properties, 290
- Action Component Reference, 291
- Action Subcomponent Reference, 323

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:

Operation	Transformers	Actions
Input	The input of a transformer is a single string.	The input is implemented by the action. An action can have multiple inputs. The inputs can be data holders.
Output	The output of a transformer is a string.	The output is implemented by the action. For example, an action can create output data holders.
Side effects	A transformer has no side effects, other than modifying the input string.	An action can have side effects, such as updating a database.

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 206.

Standard Action Properties

The following table describes standard properties of actions:

Property	Description
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 389.

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 389.
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 206.
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:

Property	Description
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 389.

Property	Description
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 206. 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:

Property	Description
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.

Property	Description
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 389.
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 389.
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.
root_element	Determines the data holder at the root of the XML branch containing the multiple-occurrence data holder. The **root_element** can be single-occurrence or multiple-occurrence.
sub_element	Determines the multiple-occurrence data holder for which the action computes an aggregate. If **sub_element** is not assigned, the action computes the aggregate of the **root_element**.
Sum	Defines an option under the **aggregation_function** property.

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:

Property	Description
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 389.
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 389.
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 206. 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 203.

Example

A source document contains the following space-separated text:

```
H E L L O
```

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 *VarLetter*. Create several `Content` anchors that retrieve the individual letters and store them in occurrences of *VarLetter*.

Then, use the **AppendListItems** action to concatenate the occurrences of *VarLetter* and store the result in the `Greeting` element. The result is:

```
<Greeting>HELLO</Greeting>
```

Online Sample

For an online sample of this action, open the project `samples\Projects\AppendListItems\AppendListItems.cmw`. The sample uses a `RepeatingGroup` to store values in a multiple-occurrence variable. It then uses as an `AppendListItems` action to concatenate the values.

AppendValues

The **AppendValues** action concatenates strings.

The following table describes the properties of the **AppendValues** action:

Property	Description
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 **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 389.
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 389.
output	Determines the data holder that stores the output. The data holder must have a simple data type.

Property	Description
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.
skip_unfound_values	Determines whether the action continues when one of the input data holders is missing. - Selected. The action continues. - Cleared. The action fails. Default is selected.

Example

A Parser has generated the following XML:

```
<Name>
  <First>Ron</First>
  <Last>Lehrer</Last>
<Name>
```

You can configure an **AppendValues** action that outputs:

```
<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:

Property	Description
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 389.
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 389.
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 206. 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 305. For more information about the precision of `xs:decimal` and `xs:float` values, see "Precision of Numerical Data" on page 191.

Example

A Parser has generated the following XML:

```
<ItemOrdered>
  <Name>Gizmo</Name>
  <Quantity>100</Quantity>
  <Price>25<Price>
</ItemOrdered>
```

You can use a **CalculateValue** action to generate the output:

```
<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.cmw`. 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:

Property	Description
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 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 **Events** view. Use the **name** property to identify the component that caused the event.

Property	Description
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 389.
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 389.
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 206. 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:

```
<DayTime>Monday morning</DayTime>
<DayTime>Monday afternoon</DayTime>
<DayTime>Tuesday morning</DayTime>
<DayTime>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 199.

Nested in this component, enter the data values.

The following table describes the properties of the **CreateList** action:

Property	Description
data_holder	Defines the multiple-occurrence data holder where the action stores the list. The data holder must have a simple data type.
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 389.
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.

Example

If the input data values are

```
Jack
Jennie
Larissa
```

the action can create the following output:

```
<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:

Property	Description
run_serializer	Determines the serializer that prepares the log message. Define a serializer in this location, or enter the name of a globally defined serializer.
output	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 323 and "Failure Handling" on page 389.

For more information about the **on_fail** property, see "Failure Handling" on page 389.

DateAddICU

The **DateAddICU** action increments a date.

The following table describes the properties of the **DateAddICU** action:

Property	Description
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_date	Defines the date to be incremented.
input_format	Defines a string or data holder that defines the date format, for example, `dd/MM/yy`. For more information, see "DateFormatICU" on page 260.
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.
num_of_days	Defines a positive or negative integer or a data holder that contains the number of days to add.
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 389.
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 389.

Property	Description
output	Determines the data holder that stores the output date.
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.

DateDiffICU

The **DateDiffICU** action computes the difference between two dates.

The following table describes the properties of the **DateDiffICU** action:

Property	Description
date1	Defines a string or data holder that defines the first date.
date2	Defines a string or data holder that defines the second date.
date_format1	Defines a string or data holder that defines the format of the first date, for example, `dd/MM/yy`. If you omit the format, the system default is used. For more information, see "DateFormatICU" on page 260.
date_format2	Defines a string or data holder that defines the format of the second date.
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 389.
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 389.

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 206. 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:

Property	Description
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 389.
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 389.
output	Determines the data holder that stores the downloaded content.

Property	Description
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 206. 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:

Property	Description
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 389.
output	Determines the data holder that stores the result. You can choose one of the following options: - OutputFile - ResultFile - StandardErrorLog 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 206. 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:

Property	Description
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`: `$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 389.
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 206. 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:

  ```
  () + - * / % == != < <= > >= && ||
  ```

- The ternary `?:` operator.
- The following methods:

  ```
  charAt
  indexOf
  ```

```
lastIndexOf
length
join
substring
toString
```

If you apply these methods to a literal having a simple data type, you must enclose the literal in parentheses, for example:

```
123.toString();                    //Wrong
(123).toString();                  //Right

"Hello, World".substring(3,7);     //Wrong
("Hello, World").substring(3,7);   //Right
```

- The following functions:

```
Math.ceil
Math.floor
Math.max
Math.min
Math.pow
Math.sqrt
parseFloat
parseInt
```

The JavaScript processor does not support features such as the following:

- The unary and binary operators:

```
++ -- typeof void >> >>> << === !== ~ & | ^
```

- Assignment operators:

```
= += -= *= /= >>= >>>= <<= &= |= ^=
```

- 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.

`extra.sum(number1, number 2, ...)`

Returns the sum of the parameters.

`extra.allSame(param1, param2, ...)`

Returns true if all the parameters have the same value.

`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 `USPresidents`, `lookup.USPresidents(1)` returns `George Washington`.

For more information, see "LookupTransformer" on page 272.

`extra.formatDate(date, input_format, output_format)`

Formats a date or time. For more information, see "DateFormatICU" on page 260.

`extra.substitute(text, oldSubstring, newSubstring, useRegex)`

Replaces all instances of a substring with another substring. If `useRegex` is true, the method interprets `oldSubstring` as a regular expression.

`extra.formatNumber(number, size_of_integer_part, number_of_decimals, sign, insert_decimal_point, unit_type)`

Formats a number. For more information, see "FormatNumber" on page 266.

`extra.insertString(text, injections_place, string_to_inject)`

Inserts a string into the text. For more information, see "InjectString" on page 271.

`extra.formatTransformationStartTime(format)`

Outputs the date and/or time at which the transformation started running. For more information, see "TransformationStartTime" on page 283.

`extra.resize(text, size, padding_character, align)`

Fits the input text to a specified size, padding or truncating as required. For more information, see "Resize" on page 279.

`extra.dateAdd(date_format, date, days_to_add)`

Increments a date by a given number of days. For more information about the `date_format`, see "DateFormatICU" on page 260.

`extra.dateAddMonths(date_format, date, months_to_add)`

Increments a date by a given number of months. For more information about the `date_format`, see "DateFormatICU" on page 260.

`extra.dateAddYears(date_format, date, years_to_add)`

Increments a date by a given number of years. For more information about the `date_format`, see "DateFormatICU" on page 260.

`extra.dateDiff(date_format1, date1, date_format2, date2)`

Computes the difference between two dates. For more information, see "DateDiffICU" on page 302.

`extra.createGuid(0)`

Generates a GUID identifier. For more information, see "CreateGuid" on page 260. You must supply a parameter value such as 0.

`extra.upper(text)`, `extra.lower(text)`, `extra.capitalize(text)`

Changes the text to all upper case, all lower case, or only the first letter capitalized. For more information, see "ChangeCase" on page 259

`extra.rtl2ltr(text)`

Reverses a string written in a right-to-left language to left-to-right. For more information, see "hebrewBidi" on page 269.

`extra.trim(text)`

Deletes leading and trailing space characters from the text. For more information, see "RemoveMarginSpace" on page 277.

ExcludeItems

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 199.

The following table describes the properties of the **ExcludeItems** action:

Property	Description
data_holder	Defines a multiple-occurrence data holder.
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 389.
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 389.
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 206. 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:

Property	Description
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 389.
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 389.
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.
source	Determines the source data holder.
source_default	Determines the default source data holder.
target	Determines the destination data holder.
transformers	Defines a sequence of transformers that modify the value. Do not assign this property if the source and destination are complex XML elements.
validators	Defines a list of validators applied to the source data. For more information, see "Validators" on page 392.

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:

Property	Description
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: - MandatoryStructureMissing. A mandatory record does not appear in the input. - 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. - XsdValidationError. 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 239.
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 206. 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:

Property	Description
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 389.
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 206. 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:

Property	Description
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 348. 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**.

Property	Description
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 345.
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 389.
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 389.
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.

RunMapplet

The **RunMapplet** action runs a mapplet. The output of **RunMapplet** is read into the data holder specified in the RunMapplet action.

Use the **RunMapplet** action to perform tasks such as data masking, data quality, data lookup, and other activities usually related to relational transformation, without the necessity to convert data to relational format and then back to hierarchical format.

Note: The RunMapplet action only can be used to call passive mapplets.

The output parameters must be specified in the same order as appears in the mapplet target ports.

The following table describes the properties of the **RunMapplet** action:

Property	Description
inputs	Specifies the input values to be passed to the mapplet. The input parameters must be specified in the same order as they appear in the mapplet source ports.
mapplet_name	Indicates the mapplet to be run. **Note:** You must first add a reference in the **References** tab for any mapplet that you want to call with the RunMapplet action. For more information, see "References" on page 29.
outputs	Specifies where to store the output values returned from the mapplet. The output parameters must be specified in the same order as they appear in the mapplet target ports. In the **OutputLocation** parameter, specify the following values: - **data_holder**: Specifies where to store the values that the mapplet returns. - **initialization**: When you test the transformation, the Data Processor transformation does not run the mapplet called by the RunMapplet action. You can specify a value to use as a temporary output string when you test the transformation during design time.

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:

Property	Description
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.
exclude_strings	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.
include_strings	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.
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.

Property	Description
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 389.
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 389.
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 206. 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 213.

RunPCWebService

The **RunPCWebService** action runs a _PowerCenter mapplet from within a Data Processor transformation. The output of **RunPCWebService** is read into the data holder specified in the **RunPCWebService** action.

Note: The **RunPCWebService** action is used to call passive mapplets.

The output parameters must be specified in the same order as appears in the mapplet target ports.

The following table describes the properties of the **RunPCWebService** action:

Property	Description
wsdl	Specify the WSDL for the web service to be run.
operationName	Indicates the operation to be run. Select one operation.
inputs	Specifies the input values to be passed to the mapplet. The input parameters must be specified in the same order as they appear in the mapplet source ports.
outputs	Specifies where to store the output values returned from the mapplet. The output parameters must be specified in the same order as they appear in the mapplet target ports.

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:

Property	Description
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 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 336. 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**.
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
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 389.
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 389.
output	Defines a data holder for the Serializer output.
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.
serializer	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 330.

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:

Property	Description
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 389.
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 389.
output	A data holder of a complex type that must match the XMap ouput type. If you omit this property, the XMap object writes to its default output definiton.
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.
xmap	Name of the XMap. You can choose the name of an existing XMap object.

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:

Property	Description
data_holder	Defines a data holder that receives 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.
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 389.
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 389.
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 206. 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.

Sort

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:

Property	Description
by_fields	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 `xs:integer`.
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 389.
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 206. 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 352.

ValidateValue

The **ValidateValue** action validates XML data according to a set of rules defined in a Validation Rules object. 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:

Property	Description
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.
errors_found	Defines a data holder that counts the number of validation rule violations in the input.
errors_output	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.
input	Defines the input data holder that the action analyzes for conformity with the Validation Rules.
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 389.
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 206.
remark	A user-defined comment that describes the purpose or action of the component.

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:

Property	Description
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 the data holder to write from.

Property	Description
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_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 389.
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 389.
output	Defines the output location. You can choose one of the following options: - OutputDataHolder. Writes to a data holder. - OutputFile. Writes to a file. Select **Synchronize** to lock the file so as to append data to the same 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 389. Default is ResultFile. For more information about these options, see "Action Subcomponent Reference" on page 323.
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 206. 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 368.

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:

Property	Description
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 the XML element to transform.
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 389.
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 389.
output	Defines a data holder to store the output.
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 206. Default is main.
remark	A user-defined comment that describes the purpose or action of the component.
xslt_file	Defines the XSLT file.

Example

Suppose that the following XML is the result of a Parser:

```
<Person>
  <First>Ron</First>
  <Last>Lehrer</Last>
</Person>
```

With an appropriate XSLT file, you can use the **XSLTMap** action to convert this to:

```
<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:

Property	Description
data_holder	Defines the output 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.
transformers	Defines a sequence of transformers that modify the stream before writing.

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:

Property	Description
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.

Property	Description
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.

CHAPTER 19

Serializers

This chapter includes the following topics:

- Serializers Overview, 325
- Serialization Anchors, 328
- Standard Serializer Properties, 330
- Serializer Component Reference, 330
- Serialization Anchor Component Reference, 331

Serializers Overview

A Serializer converts an XML or JSON 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:

- Invert the configuration of an existing Parser
- Edit the Script and inserting a **Serializer** component

You can combine also 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 or JSON 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 or JSON 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.

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:

Source text	Anchor
Name	Marker
(first and last):<tab>	Not marked as an anchor
Ron Lehrer	Content

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:

Value	Description
Full	The **Create Serializer** command copies the non-XML text to the Serializer configuration. This is the default behavior.
Outline	The **Create Serializer** command copies only the delimiters of the non-XML text to the Serializer configuration. Under the **Outline** option, you can select the **use_markers** option. This causes the Create Serializer command to copy the content of the **Marker** anchors but only the delimiters of other non-XML text.

The following table illustrates the results of the **serialization_mode** settings:

serialization_mode	Behavior	Sample Serializer Output
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

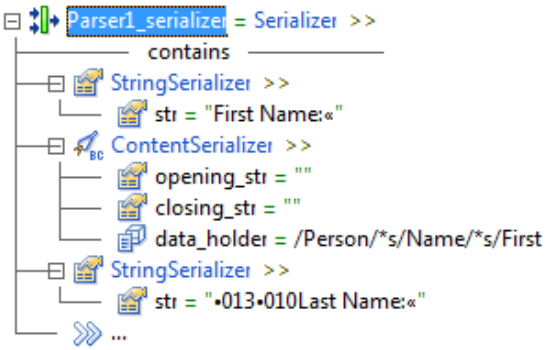
Serialization anchors are the main components you use in a Serializer. These are analogous to the anchors that are used in a Parser, except that they work in the opposite direction. Anchors read data from locations in the source document and write the data to XML. Serialization anchors read XML data 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:



The first `StringSerializer` instructs the Serializer to write the following text in the output document:

```
First Name:<tab>
```

The `ContentSerializer` writes the value of the `Person/Name/First` element to the output.

The second `StringSerializer` writes the string:

```
<newline>Last Name:<tab>
```

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<newline>Last Name<tab>
```

The display of this text is:

```
First Name:     Ron
Last Name:
```

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:

Property	Description
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 408.
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 389.
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 389.
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:

Property	Description
default_transformers	Defines a list of transformers that the **Serializer** applies to all serialized data.
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 408.
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 389.
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 352.
target	Defines a data holder that contains the result of the serialization. For more information, see "Overview of Locators, Keys, and Indexing" on page 352.
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.

Serialization Anchor Component Reference

Serialization anchors in a **Serializer** identify and manipulate data in the source document.

AlternativeSerializers

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:

Property	Description
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 389.
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 389.
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:

Property	Description
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. The 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 389.
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 389.
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:

Property	Description
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 408.
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 389.
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 389.
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 Default is StringSerializer.

Property	Description

separator_position	Defines the position of the **separator** relative to the sections. You can choose one of the following options:
	- after. Writes a separator after each section, including the first sections. For example,
	`1|2|3|4|`
	- around. Writes separators before and after each section, including the first sections. For example,
	`|1|2|3|4|`
	- before. Write a separator before each section, including the first sections. For example,
	`|1|2|3|4`
	- between. Writes a separator between the successive sections, but not before the first section and not after the last section. For example,
	`1|2|3|4`
	Default is before.

using_placeholders	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:
	- always. Always writes the separator of a missing section. For example,
	`|1||3|`
	- never. Never writes the separator of a missing section. For example,
	`|1|3`
	- when necessary. Always writes the separator of a missing internal section. Never writes the separator of a missing terminal section. For example,
	`|1||3`
	Default is when necessary.

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 `always`. 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 `never`. 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:

Property	Description
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 389.
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 389.
remark	A user-defined comment that describes the purpose or action of the component.

Property	Description
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 247. 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:

```
<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:

Property	Description
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 408.

Property	Description
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 389.
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 389.
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 388.
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 388.

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 199.

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:

Property	Description
count	Defines the number of iterations to run. If this property is blank, the iterations continue until the input is exhausted.
current_iteration	Defines a data holder where the **RepeatingGroupSerializer** outputs the number of the current iteration. You can use a **ContentSerializer** to write the number to 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.

Property	Description
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 408.
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 389.
on_iteration_fail	Determines the action when an 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 **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 389.
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 389.
remark	A user-defined comment that describes the purpose or action of the component.
separator	Defines a serialization anchor that defines the separator string. You can choose one of the following options: - AlternativeSerializers - ContentSerializer - EmbeddedSerializer - GroupSerializer - RepeatingGroupSerializer - StringSerializer - User-defined Serializer Default is blank.

Property	Description

separator_position	Defines the position of the **separator** relative to the sections. You can choose one of the following options:
	- after. Writes a separator after each section, including the first sections. For example,
	`1|2|3|4|`
	- around. Writes separators before and after each section, including the first sections. For example,
	`|1|2|3|4|`
	- before. Write a separator before each section, including the first sections. For example,
	`|1|2|3|4`
	- between. Writes a separator between the successive sections, but not before the first section and not after the last section. For example,
	`1|2|3|4`
	Default is before.

skip_failed_iterations	Determines whether failed iterations are skipped. You can choose one of the following options: - Selected. **RepeatingGroup** skips over a failed iteration and proceeds with the next iteration. If an iteration succeeds, the **RepeatingGroup** succeeds. - Cleared. **RepeatingGroup** fails if any iteration fails. The **skip_failed_iterations** property has an effect only if **separator** is defined. Default is selected.
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 388.
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 388.

Example

The XML input contains the following structure:

```
<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:

Property	Description
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 389.
remark	A user-defined comment that describes the purpose or action of the component.
str	Defines the string that the Serializer writes to the output.

CHAPTER 20

Mappers

This chapter includes the following topics:

- Creating a Mapper, 342
- Components Nested within a Mapper, 342
- Mapper Example, 343
- Standard Mapper Properties, 344
- Mapper Component Reference, 345
- Mapper Anchor Component Reference, 346

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 346.
- 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>
```

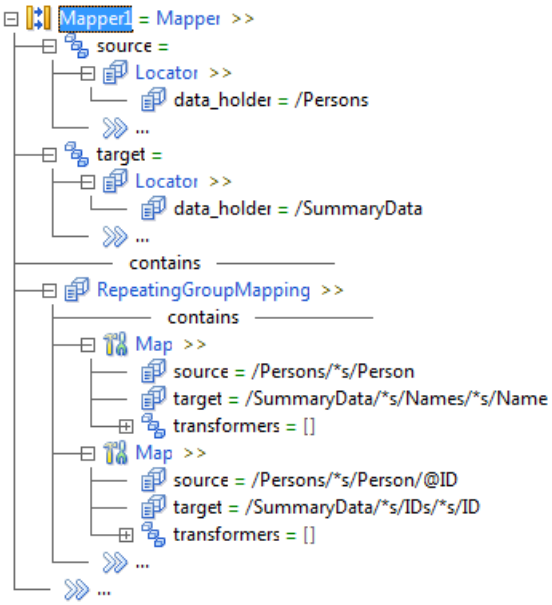
Output XML

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:



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:

Property	Description
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 408.

Property	Description
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 389.
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 389.
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

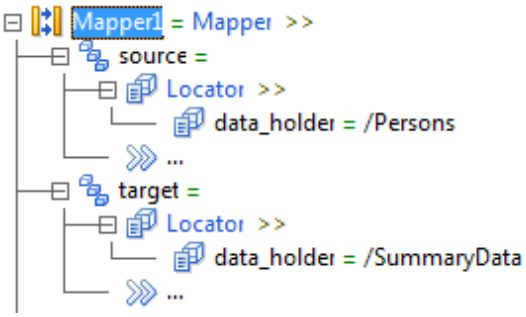
A **Mapper** performs XML-to-XML transformations. It converts a source XML document to an output document that has a different XML structure.

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

Property	Description
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 408.

Property	Description
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 389.
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 defined in the schema. For example, if the top-level element of the schema is `Person`, but the XML input nests `Person` in an element called `InputWrapper`, set **root_tag** to `InputWrapper`.
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 352.
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 352.
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. - Strict. Enforces the schema strictly. Default is Partial.

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 Anchor Component Reference

Mapper anchors in a Mapper identify and manipulate data in an XML document.

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:

Property	Description
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 389.
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 389.
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`.

EmbeddedMapper

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:

Property	Description
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 389.
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 389.
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 247. 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:

```
<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:

Property	Description
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 408.
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 389.
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 389.
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 352.
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 352.

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 199.

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:

Property	Description
count	Defines the number of iterations to run. If this property is blank, the iterations continue until the input is exhausted.
current_iteration	Defines a data holder where the **RepeatingGroupMapping** outputs the number of the current iteration.
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 408.
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 389.

Property	Description
on_iteration_fail	Determines the action when an 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 **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 389.
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 389.
remark	A user-defined comment that describes the purpose or action of the component.
skip_failed_iterations	Determines whether failed iterations are skipped. You can choose one of the following options: - Selected. **RepeatingGroup** skips over a failed iteration and proceeds with the next iteration. If an iteration succeeds, the **RepeatingGroup** succeeds. - Cleared. **RepeatingGroup** fails if any iteration fails. The **skip_failed_iterations** property has an effect only if **separator** is defined. Default is selected.
source	Defines a data holder that contains the source for the mapping. For more information, see "Overview of Locators, Keys, and Indexing" on page 352.
target	Defines a data holder that contains the result of the mapping. For more information, see "Overview of Locators, Keys, and Indexing" on page 352.

Example

For more information, including an example of a **RepeatingGroupMapping**, see the "Mapper Example" on page 343.

CHAPTER 21

Locators, Keys, and Indexing

This chapter includes the following topics:

- Overview of Locators, Keys, and Indexing, 352
- Example of Locators, 353
- Example of Indexing by Key, 354
- Source and Target Properties, 357
- Standard Locator and Key Properties, 363
- Locator and Key Component Reference, 363

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:

```
<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:

```
<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:

The output is incorrect:

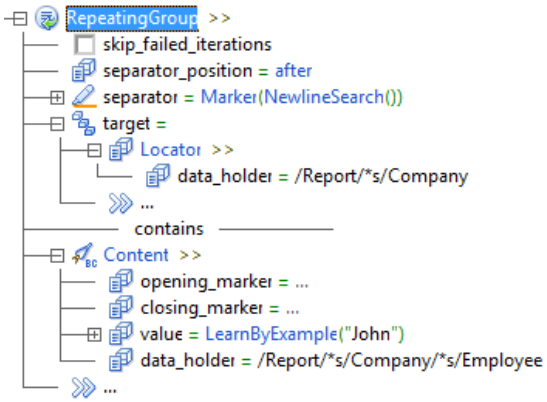
```
<Report>
  <Company>
```

```
      <Employee>John</Employee>
      <Employee>Marie</Employee>
    </Company>
  </Report>
```

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.

```
<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.

```
<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.

```
ParentKey = Key >>
    recurring_element = /Report/*s/Parents/*s/Parent
    unique_fields =
        /Report/*s/Parents/*s/Parent/@id
        ...
HobbyKey = Key >>
    recurring_element = /Hobbies/*s/Hobby
    unique_fields =
        /Hobbies/*s/Hobby/@name
        ...
```

The Script then defines a `Mapper` with the following configuration:

```
HobbiesMapper = Mapper >>
    source =
        Locator >>
            data_holder = /Report
        ...
    target =
        Locator >>
            data_holder = /Hobbies
        ...
    contains
    RepeatingGroupMapping >>
        source =
            Locator >>
                data_holder = /Report/*s/Children/*s/Child
            LocatorByKey >>
                key = /Report/*s/Parents/*s/Parent[@id=$1]
                params = |/Report/*s/Children/*s/Child/@parentID]
            ...
        target =
            Locator >>
                data_holder = /Hobbies/*s/Hobby/*s/Person
            LocatorByKey >>
                key = /Hobbies/*s/Hobby[@name=$1]
                params = |/Report/*s/Children/*s/Child/@hobby]
            ...
        contains
        Map >>
            source = /Report/*s/Children/*s/Child/@name
            target = /Hobbies/*s/Hobby/*s/Person/@firstName
            transformers = []
        Map >>
            source = /Report/*s/Parents/*s/Parent/@lastName
            target = /Hobbies/*s/Hobby/*s/Person/@lastName
            transformers = []
        ...
    ...
```

The components of the `Mapper` perform the following functions:

- The `source` property of the `RepeatingGroupMapping` 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 `RepeatingGroupMapping` 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:

Source	Description
Locator	Identifies a single-occurrence or multiple-occurrence data holder. In the latter case, each iteration accesses the next occurrence, in sequence.
LocatorByKey	Identifies an occurrence of a multiple-occurrence data holder by using a key.
LocatorByOccurence	Identifies an occurrence of a multiple-occurrence data holder by number.

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 359.
- 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 `Group` 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

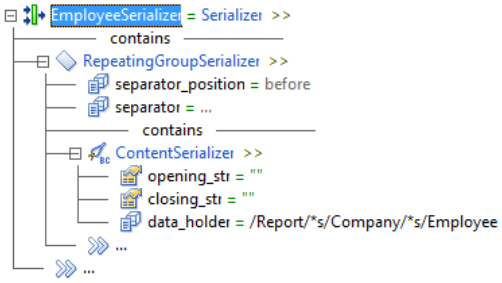
Suppose that the input schema of a serializer requires the following structure:

```
<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.

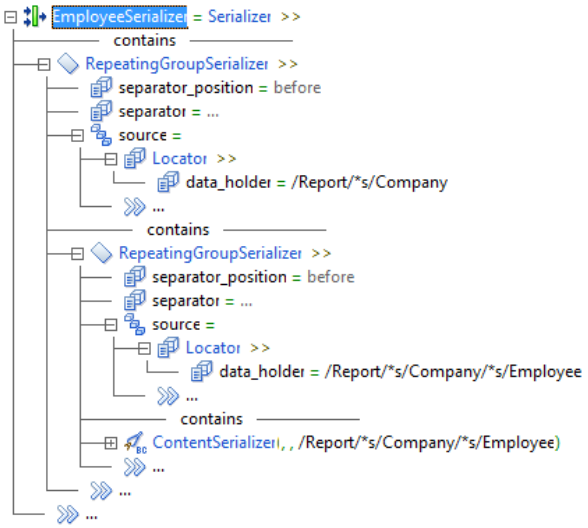


This does not work correctly. By default, each iteration selects a new instance of `Employee` within the same `Company`. The result is the following output:

```
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.

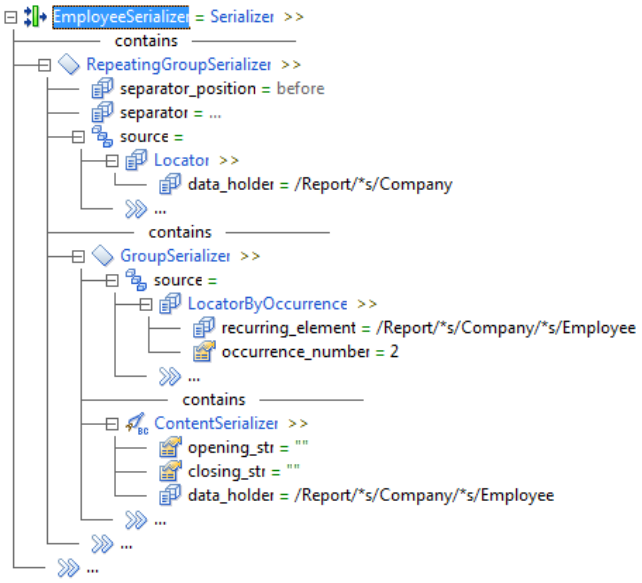


Each iteration of the outer `RepeatingGroupSerializer` processes a different occurrence of `Company`. Each iteration of the nested `RepeatingGroupSerializer` processes a different occurrence of `Employee`. The result is the desired output.

Alternatively, suppose you want to iterate only over the second `Employee` element in each `Company`. The desired output is:

```
Leslie
Larry
```

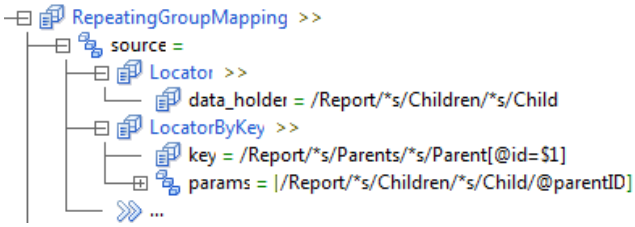
You can do this by configuring a single `RepeatingGroupSerializer`, whose source is `Company`. This causes each iteration to access the next instance of `Company`. Within the iteration, you can configure a `GroupSerializer`, whose `source` property uses a `LocatorByOccurrence` to select the second `Employee`. This generates the desired output.



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`.



For more information, see "Example of Indexing by Key" on page 354.

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:

Target	Description
Locator	Identifies a single-occurrence or multiple-occurrence data holder. In the latter case, each iteration creates a new occurrence.
LocatorByKey	Identifies an occurrence of a multiple-occurrence data holder by an indexing key. If the occurrence does not yet exist, it is created.
LocatorByOccurence	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 **LocatorByOccurrence** specifies the tenth occurrence, occurrences 5-9 are also created, but left empty.

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 362.
- 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.

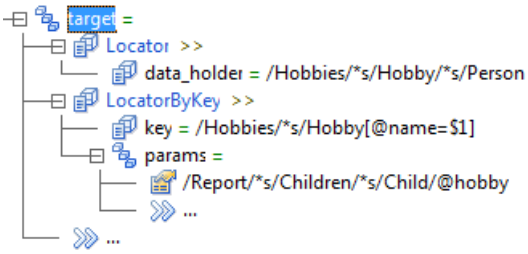
It is also possible to use the property to identify where a component should obtain input. For example, suppose that a `GroupSerializer` contains an action that computes data and stores it in a variable. The `GroupSerializer` then activates a `ContentSerializer` that writes the variable to the output. 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 353.

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`:



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 354.

Standard Locator and Key Properties

The following table describes standard properties that are used in the locator and key components:

Property	Description
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 389.
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:

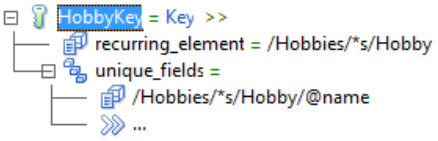
Property	Description
disabled	Determines whether the Script ignores the component and all of the child components. Use this property to test, debug, and modify a Script. You can choose one of the following options: - Selected. The Script ignores the component. - Cleared. The Script applies the component. The default is cleared.
recurring_element	Defines a multiple-occurrence element whose occurrences are identified by the key.
remark	A user-defined comment that describes the purpose or action of the component.
unique_fields	Defines the key.

Example

The `Example of Indexing by Key` defines a key for the `Hobby` element in the following structure:

```
<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:

```
<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`.

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:

```
<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:

Property	Description
data_holder	Defines the data holder that the **Locator** component identifies.
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 389.
remark	A user-defined comment that describes the purpose or action of the component.

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:

Property	Description
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.
key	Defines the XPath predicate representation of the key. For example, if you have defined `Hobbies/Hobby/@name` as a **Key**, select `Hobbies/Hobby[@name=$1]`. This property is required.
optional	Determines whether a component failure causes the parent component to fail. You can choose one of the following options: - Selected. Component failure does not cause the parent component to fail. - Cleared. Component failure causes the parent component to fail. Default is cleared. For more information about component failure, see "Failure Handling" on page 389.
params	Defines the values of the parameters in the XPath predicate. Each value has a dollar sign ($) and an integer that represents the position of the parameter in the list of parameters. This property is required.
remark	A user-defined comment that describes the purpose or action of the component.

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:

Property	Description
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.
occurrence_number	Defines the number of the occurrence.
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 389.
recurring_element	Defines the data holder that the component identifies.
remark	A user-defined comment that describes the purpose or action of the component.

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 22

Streamers

This chapter includes the following topics:

- Streamers Overview, 368
- Text Streamers, 369
- XML Streamers, 373
- Standard Streamer Properties, 375
- Streamer Component Reference, 375
- Streamer Subcomponent Reference, 384

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 can have a buffer input or file input.

A Streamer offers the following advantages:

- The transformation parses each source segment when it is available, instead of waiting until it receives entire source.
- The transformation has reduced memory requirements.

For example, a Streamer input stream might contain stock market transaction data. The stream transmits to a server continually over the course of the entire trading day. A Script with a Streamer processes each transaction when it arrives, instead of waiting until the end of the day.

In another example, you receive a large source file over an FTP connection. By using a Streamer, the Script can process 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. Define the **Streamer** or **XmlStreamer** component at the global level of the Script, and set it as the startup component of the transformation. The Streamer functions by splitting the 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.

Segment	Type	Opening Marker	Closing Marker
Header	Simple	[0-9][0-9]-[0-9][0-9]-[0-9][0-9]/	ENDHEAD/
Repeating	Simple	TRANS [0-9][0-9]:[0-9][0-9] [0-9]+/	none
Footer	Simple	END/	none

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 version="1.0" encoding="windows-1252"?>
<header>...</header>

<?xml version="1.0" encoding="windows-1252"?>
<repeating_segment>...</repeating_segment>

<?xml version="1.0" encoding="windows-1252"?>
<repeating_segment>...</repeating_segment>

<?xml version="1.0" encoding="windows-1252"?>
<footer>...</footer>
```

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:

```
<MyRoot>
  <?xml version="1.0" encoding="windows-1252"?>
  <header>...</header>

  <?xml version="1.0" encoding="windows-1252"?>
  <repeating_segment>...</repeating_segment>

  <?xml version="1.0" encoding="windows-1252"?>
  <repeating_segment>...</repeating_segment>

  <?xml version="1.0" encoding="windows-1252"?>
  <footer>...</footer>
</MyRoot>
```

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

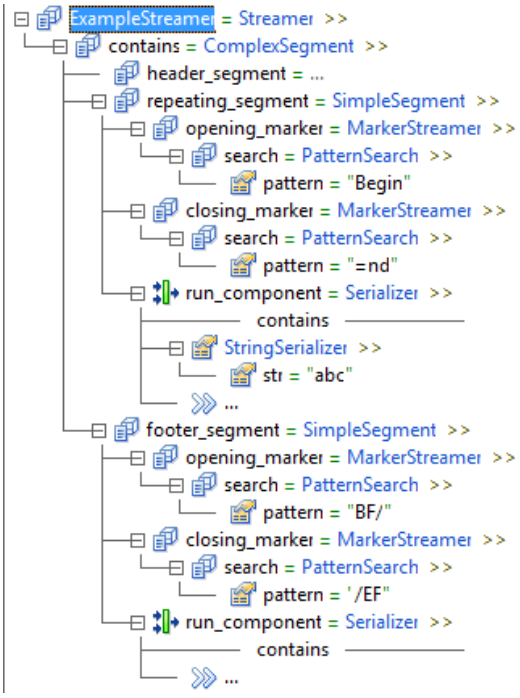
1. Analyze the source structure and identify the segment types.
2. Create or open a Script.
3. In the Script, configure a transformation such as a Parser, Mapper, or Serializer that can process each type of simple segment.
4. In the same Script, configure a **Streamer** component.

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.

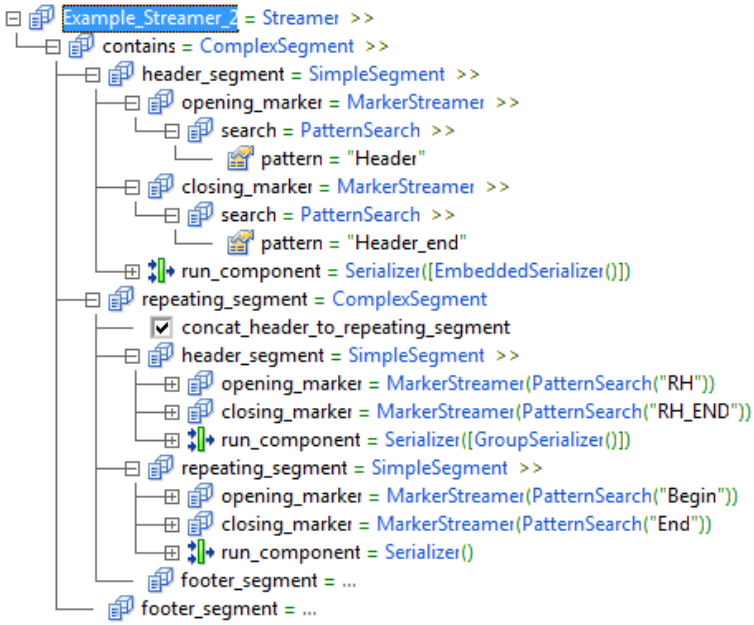
The Streamer passes the header and repeating segments to a Parser called `body_p`. It passes the footer to a Parser called `foot_p`.



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 `body_p`.



XML Streamers

An `XmlStreamer` component splits a large XML document into smaller portions. The `XmlStreamer` divides the XML source into header, body, and footer 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, 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:

```
<stream>
  <headerline1>MainHeader</headerline1>
  <substreams>
    <substream>
      <subheaderline1>SubHeader</subheaderline1>
      <segments>
```

```
          <segment1>Segment1A</segment1>
          <segment1>Segment1B</segment1>
          <segment2>Segment2A</segment2>
          <segment1>Segment1C</segment1>
          <segment2>Segment2B</segment2>
        </segments>
        <subfooterline1>SubFooter</subfooterline1>
      </substream>
      <substream>...</substream>
      <substream>...</substream>
    </substreams>
    <footerline1>MainFooter</footerline1>
  </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>
```

The footer is everything that follows the last `substream`:

```
  </substreams>
  <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:

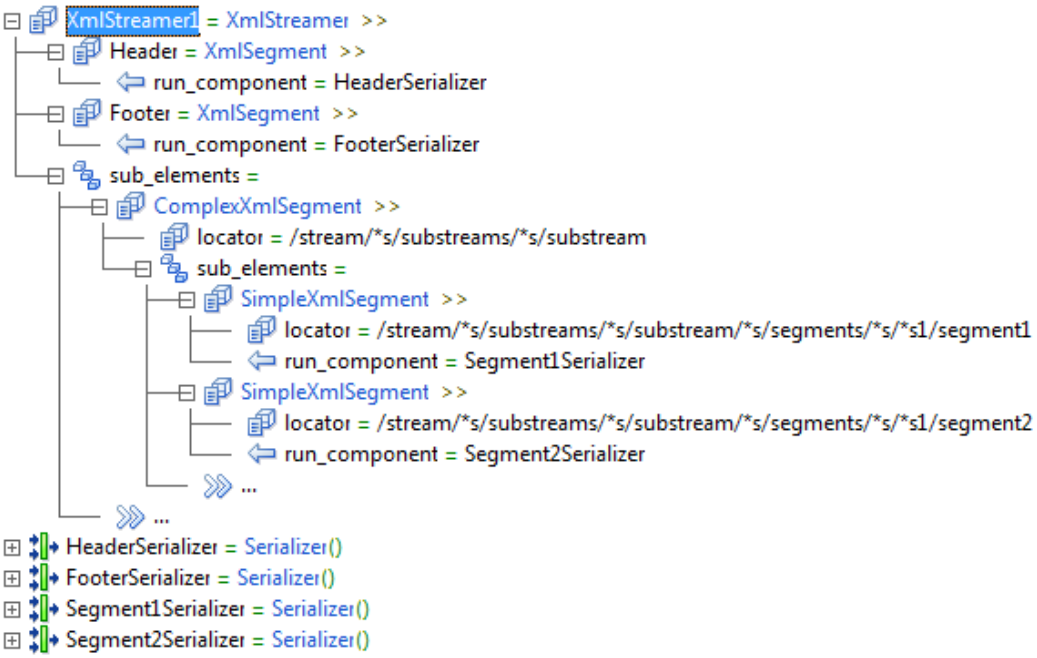
```
<stream>
  <headerline1>MainHeader</headerline1>
  <substreams>
  </substreams>
</stream>
```

You can configure the `XmlStreamer` 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.

Note: The header segment elements are available when processing the body elements. The footer elements, however, are not yet available when processing body elements. Footer elements are only processed after the transformation finishes reading body elements.

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 `XmlStreamer` 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.



Note: Even through the footer run component appears before the body elements in this example, footer elements are only processed after the transformation finishes reading body elements.

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:

Property	Description
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.

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:

Property	Description
concat_header_to_repeating_segment	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 370. 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:

Property	Description
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 382. Default is XmlSegment.

Property	Description
Header	Defines how to process the header of the **ComplexXmlSegment**. For more information, see "XmlSegment" on page 382. 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**.

JsonStreamer

The **JsonStreamer** accepts a very large JSON file input and splits the input into segments. It passes each type of segment to a predefined transformation such as a Parser, Mapper, or Serializer.

The **JsonStreamer** must be defined at the global level of the Script and it must be the startup component of the transformation.

Note: The number of segments is automatically determined according to the precision of the input port.

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

Property	Description
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 387.

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:

Property	Description
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.
remark	A user-defined comment that describes the purpose or action of the component.
search	Defines how the **MarkerStreamer** finds text. You can choose one of the following options: - NewlineSearch. Searches for a newline character. - OffsetSearch. Skips a predefined number of characters following the preceding reference point. - PatternSearch. Searches for a regular expression. - TextSearch. Searches for an explicit string.

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.

Marking of Opening Marker	Marking of Closing Marker	Segment Passed to the Transformation
full	full	`...data...`
full	begin	`...data...`
full	end	`...data...END`
begin	full	`BEGIN...data...`
begin	begin	`BEGIN...data...`
begin	end	`BEGIN...data...END`
end	full	`...data...`
end	begin	`...data...`
end	end	`...data...END`

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 275.

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

Property	Description
closing_marker	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.
count	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.

Property	Description
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 387.

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:

Property	Description
count	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.
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.
locator	Defines a data holder.

Property	Description
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. - 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 384. Default is DoNothingModifier.
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 387.

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:

Property	Description
contains	Defines the overall structure of the source. Default is ComplexSegment.
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.

Property	Description
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 370.

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 194.

You can define a **StreamerVariable** only at the global level of the Script.

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

Property	Description
initialization	An initial value for the **StreamerVariable**, assigned when the transformation starts. Select **InitialValue** and enter the value.
val_type	Defines the data type that the variable can store. Assign a simple type such as `xs:string` or `xs:integer`. Streamer variables cannot have complex or multiple-occurrence types. Default is `xs:string`.

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:

Property	Description
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 384. 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 387.

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:

Property	Description
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 382. Default is XmlSegment.
Header	Defines how to process the header of the **ComplexXmlSegment**. For more information, see "XmlSegment" on page 382. 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:

Property	Description
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.

```
<stream>
  <headerline1>...</headerline1>
  <segments>
    <segment1>...</segment1>
    <segment1>...</segment1>
    <segment1>...</segment1>
  </segments>
  <footerline1>...</footerline1>
</stream>
```

You can configure an **XmlStreamer** that returns the following header, which is not well-formed XML:

```
<stream>
  <headerline1>...</headerline1>
  <segments>
```

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:

```
<stream>
  <headerline1>...</headerline1>
  <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:

Property	Description
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:

Property	Description
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:

```
<stream>
  <headerline1>...</headerline1>
  <substreams>
    <substream>
      <subheaderline1>...</subheaderline1>
```

```
        <segments>
          <segment1>...</segment1>
          <segment1>...</segment1>
          <segment1>...</segment1>
        </segments>
        <subfooterline1>...</subfooterline1>
      </substream>
      <substream>...</substream>
      <substream>...</substream>
    </substreams>
    <footerline1>...</footerline1>
  </stream>
```

Suppose you configure an **XmlStreamer** that returns the following header, which is not well-formed XML:

```
<substream>
  <subheaderline1>...</subheaderline1>
  <segments>
```

If you apply **WellFormedModifier** to this header, the modifier adds the closing tags. The result is the following well-formed segment:

```
<substream>
  <subheaderline1>...</subheaderline1>
  <segments>
  </segments>
</substream>
```

Now suppose that you configure **WellFormedModifier** to add `<stream>` as a root element. The result is:

```
<stream>
  <substreams>
    <substream>
      <subheaderline1>...</subheaderline1>
      <segments>
      </segments>
    <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:

Property	Description
output	Defines 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. For more information, see "Failure Handling" on page 389. For more information about these options, see "Action Subcomponent Reference" on page 323. Default is ResultFile.

CHAPTER 23

Validators, Notifications, and Failure Handling

This chapter includes the following topics:

- Overview of Validators, Notifiers, and Failure Handling, 388
- Failure Handling, 389
- Validators, 392
- Standard Validator Properties, 392
- Validator Component Reference, 393
- Notifications, 408
- Notification Component Reference, 409

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

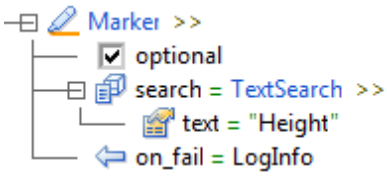
To define the user log output, assign the `on_fail` property of the appropriate transformation components. The following components have an `on_fail` property:

- 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 300.
- `NotifyFailure`. Triggers a notification.

The following example illustrates a `Marker` anchor with a `LogInfo` configuration:



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
```

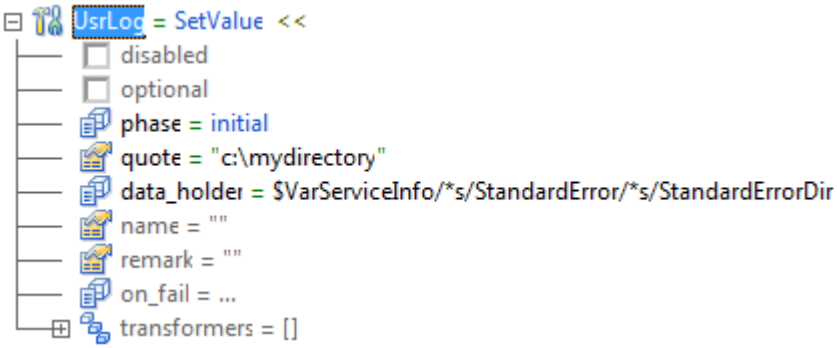
By default, each execution of a transformation generates a user log having a unique name:

```
<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.

Variable	Description
VarServiceInfo > **StandardError** > *StandardErrorDir*	Directory path of the user log.
VarServiceInfo > **StandardError** > *StandardErrorName*	File name of the user log.

In the following example, a **SetValue** action sets the user-log directory to `c:\mydirectory`.



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 408.

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 319.

Standard Validator Properties

The following table describes standard properties of validators:

Property	Description
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 408. 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 389.
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.

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:

Property	Description
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.

Property	Description
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 408. 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 389.
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:

Property	Description
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 389.
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.

Property	Description
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.
param3	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.
param4	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.
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:

Property	Description
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.

Property	Description
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 408. 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 389.
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:

Property	Description
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 408. 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 389.
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.

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:

Property	Description
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 408. 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 389.
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.

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:

Property	Description
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 408. 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 389.

Property	Description
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.
value	Defines the maximum value of the number.

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:

Property	Description
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 minimum 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 408. 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 389.

Property	Description
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.

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:

Property	Description
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 408. 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 389.
remark	A user-defined comment that describes the purpose or action of the component.

Property	Description
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.
value	Defines the minimum value of the number.

NumberEquals

The **NumberEquals** validator tests whether a number is equal to a specified value.

The following table describes the properties of the **NumberEquals** validator:

Property	Description
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 408. 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 389.
remark	A user-defined comment that describes the purpose or action of the component.

Property	Description
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.
value	Defines the value of the number.

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:

Property	Description
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.
expression	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`: `$0 == "Ron Lehrer"`
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 408. 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 389.

Property	Description
params	Defines a list of data holders that contain parameters for use in the expression.
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.

ValidateByPattern

The **ValidateByPattern** validator tests whether a string matches a regular expression. For more information, see "RegularExpression" on page 275.

The following table describes the properties of the **ValidateByPattern** validator:

Property	Description
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.
expression	Defines a regular expression.
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 408. 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 389.

Property	Description
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.

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:

Property	Description
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 408. 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 389.
remark	A user-defined comment that describes the purpose or action of the component.

Property	Description
run_transformers	Defines a list of transformers.
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.

ValidateByType

The **ValidateByType** validator tests whether its input conforms to a specified data type.

The following table describes the properties of the **ValidateByType** validator:

Property	Description
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 408. 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 389.
remark	A user-defined comment that describes the purpose or action of the component.

Property	Description
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.
val_type	Defines a data type. Select a standard type or a type that is defined in the project schemas.

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 260.

The following table describes the properties of the **ValidateDate** validator:

Property	Description
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.
format_string	Defines an ICU date 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.
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 408. 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 389.

Property	Description
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.

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:

Property	Description
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 408. 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 389.

Property	Description
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.

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:

Notification	Description
MandatoryStructureMissing	A mandatory record does not appear in the input.
MismatchIDs	The record and subelement IDs partially match. For example, there are two record identifiers, and only one of them matches.
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. For example, the subelements define a sequence `ABC`, but the input contains `ACB`.
UnexpectedRecord	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`.
UnrecognizedRecord	No subelement matches any of the record identifiers.
XsdValidationError	The input does not match the requirements of the schema.

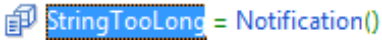
Notification Component Reference

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.

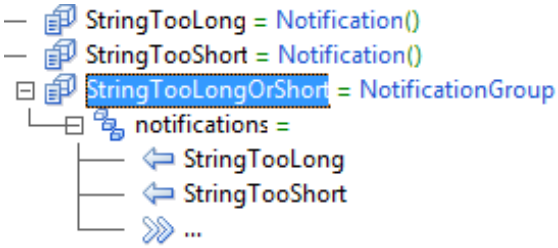
The following figure shows a notification called `StringTooLong`:



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 `StringTooLongOrShort`:



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.

Property	Description
notifications	Defines a list of notifications.

NotificationHandler

The **NotificationHandler** component defines a list of actions to take for a specified notification.

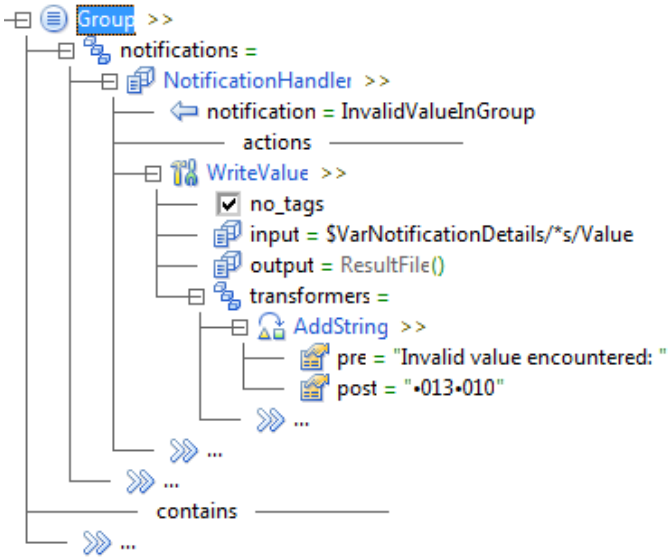
Insert the **NotificationHandler** component in locations such as the **notifications** property of a **Group** or **RepeatingGroup**. Within the component, you can insert a **WriteValue** action that stores a message in a data holder.

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 195.

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

Property	Description
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.

Property	Description
notify	Defines the notification to trigger. Select a predefined notification or a notification name that is defined in a **Notification** or **NotificationGroup** component.
value	Defines the value of the *VarNotificationDetails/Value* variable. A **NotificationHandler** can include the value in its output.

CHAPTER 24

Validation Rules

This chapter includes the following topics:

- Validation Rules Overview, 412
- Validation Rules Element Reference, 413
- Edit the Validation Rules in an External Editor, 419
- Create a Validation Rules Object, 419
- Import a Data Transformation Service with Validation Rules, 419

Validation Rules Overview

A Data Processor transformation uses Validation Rules to check the input or output data for the transformation. You can use a Validation Rules object to validate XML data according to a set of user defined rules. If the data violates the rules, the action generates an XML validation report. When you create a Validation Rules object, you can provide a sample file to test the Validation Rules object.

Validation Rules check the input or output elements for a Data Processor transformation. Use the Validation Rules editor to create, define, edit, and manage rules.

After you create a Validation Rules object, you add Validation Rules elements. You define each element in the editor. The editor adds the elements to the Validation Rules hierarchy.

The root level of the hierarchy contains a description for the Validation Rules object, and a reference to the sample XML file that you can use to debug the Validation Rules object. The Validation Rules hierarchy contains the Lookups and Rules folders. A Lookup element defines a lookup table that contains codes and translations. You can add one or more Lookup elements to the Lookup folder.

The Rule element defines a validation rule. If the Rule evaluates to false, the Validation Rules object reports an error. You can add one or more Rule elements to the Rules folder.

You can nest the following elements within a Rule element:

Variable Element

A Variable element defines a variable that has a simple data type. The element contains an XPath expression that evaluates to the value of the Variable.

Assert Element

The Assert element defines the logic of a rule. The element contains an XPath expression. If the expression is false, the Rule reports a validation error.

List Element

The List element defines a complex variable that contains a list of values.

Trace Element

The Trace element adds the value of an XPath expression to the events file.

After you create an element, you define the attributes for the element. The attributes define the logic for that element.

You can copy and paste elements into the Validation Rules editor. You can also edit the Validation Rules object in an external editor and add, edit, or delete elements in XML.

You can import a Data Transformation service with any number of VRL files. You can copy all or part of a VRL file into the Validation Rules editor. You can open the VRL file in an external editor and copy elements, and then paste them into the Validation Rules editor hierarchy.

You can call the Validation Rules object from within a Data Processor transformation Script component with the **ValidateValue** action.

Validation Rules Element Reference

The top level of a validation rule hierarchy contains a Rule or Lookup element. Within a Rule element, you can nest the Assert, List, Trace, and Variable elements.

Assert Element Attributes

The Assert element defines the logic of a rule. The element contains an XPath expression. If the expression is false, the rule reports a validation error.

A Rule must contain at least one Assert element.

The following table describes the properties of the Assert element:

Property	Description
Additional Data	Optional. An XPath expression that can be evaluated and inserted into the error report.
Code	Optional. A string attribute that identifies the Assert element. If no Code is specified, the Code is taken from the parent Rule.
Description	Optional. A description of the Assert element. If no description is specified, the description is taken from the parent Rule.
Location	Optional. A string value that can be appended to the XPath expression equivalent to the node selected by the parent Rule. The expression is then inserted into the error report.
Rule Description	A read-only description of the Rule that the element is nested under.
XPath	Mandatory. An XPath Boolean expression that expresses the logic of the rule.

List Element Attributes

The List element defines a complex variable containing a list of values.

The following table describes the properties of the List element:

Property	Description
Append	A Boolean attribute that determines what happens if the list has the same name as an existing list. If enabled, the new values are added to the existing list. If disabled, the existing list is deleted and a new list is created. The default value is enabled.
Function	Optional. An XPath expression that is evaluated relative to each of the selected nodes. The value of the expression is added to the list.
Name	Mandatory. The name of the list. Use this name to reference the List in XPath expressions of elements nested within the parent Rule.
Rule Description	A read-only description of the Rule that the element is nested under.
Select	Mandatory. An XPath expression that selects nodes for calculating the items in the list.

Lookup Element Attributes

The Lookup element defines a lookup table containing codes and translations.

The following table describes the properties of the Lookup element:

Property	Description
File	Mandatory. The file that contains the lookup table.
Name	Mandatory. The name of the lookup table.

Lookup File

The following example shows a sample Lookup file:

```
<LookupTable xmlns="http://www.Itemfield.com/Engine/V4/lookupTable" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
        <Entry key="a" value="1"/>
        <Entry key="b" value="2"/>
        <Entry key="c" value="3"/>
</LookupTable>
```

The lookup table format is identical to an `XMLLookupTable` used in the `LookupTransformer` component. For more information, see "LookupTransformer" on page 272.

You can use the `dt:lookup()` function in an XPath expression to look up a value in the lookup table.

Rule Element Attributes

The Rule element defines a validation rule. If the Rule evaluates to false, validation rule reports an error.

The following table describes the properties of the Rule element:

Property	Description
Code	Mandatory. An identifier for the Rule. Must be a single word.
Description	Mandatory. A description of the Rule in free-text, of any length.
Enabled	Determines whether the Rule is enabled or disabled. The default value is enabled.
Run All Asserts	Determines whether the Asserts that are nested under the Rule are run. By default the property is enabled, so the Rule processes all the Asserts. If disabled, the rule stops processing Asserts after one of the Asserts fails.
Select	Mandatory. An XPath expression that selects the node or set of nodes that the Rule applies to.

Trace Element Attributes

The Trace element adds the value of an XPath expression to the error report. The element contains the XPath expression.

The following table describes the properties of the Trace element:

Property	Description
Description	Mandatory. A description of the Trace element.
Enabled	Determines whether the Trace is enabled or disabled. The default value is enabled.
Rule Description	A read-only description of the Rule that the element is nested under.
XPath	Mandatory. An XPath expression that can be evaluated and inserted into the error report.

Variable Element Attributes

A Variable element defines a variable that has a simple data type. The element contains an XPath expression. The value of Variable is the value of the expression.

The following table describes the properties of the Variable element:

Property	Description
Name	Mandatory. The name of the Variable. Use this name to reference the Variable in XPath expressions of elements nested within the parent Rule.
Rule Description	A read-only description of the Rule that the element is nested under.
XPath	Mandatory. An XPath expression that calculates or refers to the nodes that the Variable applies to.

XPath Editor

Some elements contain XPath expressions. To create the expressions, you type an XPath expression into the XPath editor.

XPath Extensions

The Rule, Assert, List, Variable, and Trace elements can contain XPath expressions. The following functions can be part of the XPath expression.

The Validation Rules object defines the following functions as extensions of XPath 1.0. The extensions belong to the **dt** namespace, which is defined as `http://validations.informatica.com`.

Function	Description

`dt:all-equal(`
`  param1[,`
`  param2,`
`  ...])`
	Returns true if all items in the list are equal. The parameters can be simple values or lists.

`dt:check-uniqueness(`
`  value1[,`
`  value2,`
`  ...])`
	Returns true if all values in the list of parameters are unique. Each value can be:
	- A simple value
	- A variable that contains a list of strings
	- A variable that contains a list of nodes where each node is a simple value.

`dt:date-add(`
`  startDate,`
`  format,`
`  numberOfDays)`
	Returns the date. If **numberOfDays** is of type **float**, the function rounds down the value to the next lower integer.

`dt:date-diff(`
`  startDate,`
`  format,`
`  endDate,`
`  format)`
	Returns the number of days between two dates.

`dt:date-format(`
`  date,`
`  inputFormat,`
`  outputFormat)`
	Converts the format of **date** from **inputFormat** to **outputFormat**.

`dt:date-valid(`
`  string-to-match,`
`  format)`
	Returns true if the string matches the specified date format.

`dt:deep-equal(`
`  element1,`
`  element2)`
	Returns true if both of the following are true:
	- **element1** and **element2** have all the same attributes, with all the same values.
	- **element1** and **element2** have all the same child elements, in the same order, and all the child elements are deep equal.

`dt:empty(` `  xpath)`	Returns true if an XPath contains no child elements.
`dt:exist(` `  xpath)`	Returns true if an XPath exists. Equivalent to **count(xpath) > 0**.

Function	Description

`dt:is-sorted-lex(`
`  ascending,`
`  param1[,`
`  param2,`
`  ...])`
	Returns true if the list items are lexicographically sorted. The first parameter is a Boolean that sets the sort direction: **true** for ascending or **false** for descending.

`dt:last-day-of-month(`
`  date,`
`  format)`
	Returns the last day of the selected month.

`dt:list-items(`
`  list,`
`  separator)`
	Returns a string of all the items in the list, separated by the **separator**. The default separator is a comma.

`dt:lookup(`
`  string,`
`  lookupName,`
`  default)`
	Looks up a string in a lookup table. Specify the string to look up, the name of the lookup table, and a default value to return if the string does not exist in the table.

`dt:min-lex(`
`  value1[,`
`  value2,`
`  ...])`
	Sorts the list of input parameters lexicographically and returns the first item in the sorted list. Each value can be:
	- A simple value
	- A variable that contains a list of strings
	- A variable that contains a list of nodes where each node is a simple value.

`dt:next-sequence()`	Returns a node-set that contains the current element and all its following siblings, until another element of the same name.

`dt:regex-match(`
`  string-to-match,`
`  regex)`
	Returns true if a string matches a regular expression.

`dt:regex-replace(`
`  inputString,`
`  patternRegex,`
`  replacementString)`
	Returns **inputString** with all instances of **patternRegex** replaced by **replacementString**. If nothing matches **patternRegex**, the **inputString** value returns with no changes.

`dt:string-replace(`
`  string1,`
`  string2,`
`  string3)`
	Replaces all instances of **string2** in **string1** with **string3**.

`dt:unadjusted-calculation-period-dates(`
`  startDate,`
`  endDate,`
`  timeUnit,`
`  timeUnitMultiplier,`
`  lookupDate,`
`  dateFormat)`
	Returns true if **lookupDate** is within the first specified time period in the specified range, where the time period is **timeUnitMultiplier** times **timeUnit** and the range is between **startDate** and **endDate**.
	- **timeUnitMultiplier** is an integer.
	- **timeUnit** is one of the following strings:
	  - **Day** or **D**
	  - **Week** or **W**
	  - **Month** or **M**
	  - **Year** or **Y**
	- **lookupDate** is a date in the range between **startDate** and **endDate**.
	- **dateFormat** defines the formats of **lookupDate**, **startDate**, and **endDate**

For more information about the date format, see "DateFormatICU" on page 260.

Example: Using dt:next-sequence()

You can use the dt:next-sequence() function to access logically hierarchical data that is not nested within the current element.

Consider the following input:

```
<Root>
  <A/>
  <B/>
  <C/>
  <A/>
</Root>
```

The XPath `/Root/A/dt:next-sequence()` returns the following node-set:

```
<A/>
<B/>
<C/>
```

In the following example, each `id` element is associated with a set of sibling elements called `name`, `quantity`, and `price`:

```
<items>
  <id>100</id>
  <name>Plate</name>
  <quantity>4</quantity>
  <price>10</price>
  <id>101</id>
  <name>Toaster</name>
  <quantity>6</quantity>
  <price>10</price>
  <id>102</id>
  <name>Knife</name>
  <quantity>10</quantity>
  <price>5</price>
</items>
```

The `name`, `quantity`, and `price` elements are all logically nested within `id`, even though they are not nested physically.

The following rule requires that `price*quantity` for each `id` be less than or equal to 50:

```
<rule code="sum1" select="/items/id" description="For each id, check that the total
price (price*quantity) does not exceed 50">
  <variable name="current">dt:next-sequence()</variable>
  <variable name="total">$current[3]*$current[4]</variable>
  <assert additionalData="$total"><![CDATA[ $total <= 50 ]]></assert>
</rule>
```

For the first `id` element, the variable `$current` is a node-set with the following value:

```
<id>100</id>
<name>Plate</name>
<quantity>4</quantity>
<price>10</price>
```

The expression `$current[3]*$current[4]` evaluates to 4*10 = 40. The rule confirms that 40 < 50.

For the subsequent `id` elements, the rule evaluates the expression in a similar way.

Edit the Validation Rules in an External Editor

You can select to edit a Validation Rules in an external editor. The editor displays the Validation Rules hierarchy in XML with any nested elements that you created. You can copy, edit, or delete elements in XML.

Note: If you select to save changes when you edit a Validation Rules object in an external editor, you save the entire transformation, rather than just the Validation Rules changes.

Create a Validation Rules Object

You can use the Validation Rules editor to create, view, and edit a Validation Rules object.

1. On the Data Processor transformation **Objects** view create a Validation Rules.
2. To open the Validation Rules, click the Validation Rules object.
3. To add an element, in the left pane of the editor, right-click the rule hierarchy and select New > <element>, where <element> stands for the type of element that you want to add.

   A blank element appears.
4. In the right pane, define the attributes for the element.
5. Add elements and assign attributes to each element as required.

   At each stage, you can right-click and perform the following operations:

   - Append a sibling element
   - Add a child element
   - Delete an element
   - Enable or disable a rule
   - Undo or redo operations
6. Save the Validation Rules object.

Import a Data Transformation Service with Validation Rules

You can import a Data Transformation service containing Validation Rules from the file system repository of the machine where you saved the service. Import a Data Transformation service .cmw file to the Model repository to create a Data Processor transformation. The Developer tool imports the transformation and validation rules with the .cmw file.

1. Click **File** > **Import**,

   The **Import** dialog box appears.
2. Select **InformaticaImport Data Transformation Service** and click **Next**.

   The **Import Data Transformation Service** page appears.
3. Browse to the service .cmw file that you want to import.

The Developer tool names the transformation according to the service file name. You can change the name.

4. Browse to a location in the Repository where you want to save the transformation, then click **Finish**.

   The Developer tool imports the transformation and validation rules with the `.cmw` file.

5. To edit the Validation Rules object, double-click the Validation Rules object in the **Object Explorer** view.

   The Validation Rules editor appears and displays the Validation Rules object hierarchy.

CHAPTER 25

Custom Script Components

This chapter includes the following topics:

- Custom Script Components Overview, 421
- Custom Component Example, 421
- Custom Component Properties, 422
- Developing a Custom Component in Java, 422
- Developing a Custom Component in C or C++, 424
- Configuring a Custom Component, 426

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:

Property	Description
KeepLineBreaks	A Boolean property. When true, the processor preserves the line-break characters in the binary data.
MaxLineLength	An integer property. Specifies the maximum length of the text lines to output.
Ignore	A string property. Tells the processor to ignore data fields beginning with the specified string.

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.

Component type	Language	Maximum Properties
Document processor	Java	4
Document processor	C or C++	5
Transformer	Java	10
Transformer	C or C++	10

Developing a Custom Component in Java

1. Create a class that implements one or more of the following interfaces:

Component Type	Type of Input	Interface
Document processor	File	CMXFileProcessor
Document processor	Buffer	CMXByteArrayProcessor

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 426.

   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:

```
public String process(
    CMXContext context,
    String in,
    String additionalFilesDir,
    CMXEventReporter reporter)
    throws Exception
```

The meaning of the parameters is as follows:

Parameter	Description
context	Input parameter. An object containing the properties that the Script passes to the component. The **parameters** method of the object returns a vector containing the property values.
in	Input parameter. The full path of the file that the component operates on.
additionalFilesDir	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.
reporter	Input parameter. An object providing the report method, which the component can use to write events to the event log.

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
```

The directory contains the following samples:

Sample	Description
FilePP.java	A document processor accepting file input.
ByteArrayPP.java	A document processor accepting buffer input.
StringTT.java	A transformer accepting string input.
ByteArrayTT.java	A transformer accepting buffer input.

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
   api/include
   ```

4. Set the linker option to add the following subdirectory:

   ```
   <INSTALL_DIR>/DataTransformation/api/lib
   ```

5. Create a module that implements the appropriate functions:

Component Type	Input Type	Interface
Document processor	File	CMXProcessFile
Document processor	Multiple files	CMXProcessMultipleFiles
Document processor	Buffer	CMXProcessBuffer
Document processor	C++ stream	CMXProcessStream
Transformer	Null-terminated string	CMXTransformBuffer
Transformer	Buffer input that is not null-terminated	CMXTransformBinaryBuffer

   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 426.

   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:

```
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:

Parameter	Description
sessionToken	Input parameter. A pointer to the current session.
params	Input parameter. A structure containing the properties that the Script passes to the component.
in_file	Input parameter. The full path of the file upon which the component should operate.
in_len	Input parameter. The length, in bytes, of the input file path.
out_file	Output parameter. The full path of a file that contains the output of the component.
out_len	Output parameter. The length, in bytes, of the output file path.
reporter	Input parameter. Provides the `report` method, which the component can use to write events to the event log.
Return value	Output parameter. 1 if successful, 0 if unsuccessful.

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:

Sample	Description
Processor.c	A document processor accepting either file or buffer input.
Transformer.c	A transformer accepting null-terminated string input.

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 <CustomPropertyName1> ofPT <DataType>
   {
       paramName = "<CustomPropertyName1>" ;
   }
   ```

   `<CustomPropertyName1>` 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:

For	ComponentType

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.

4. Save the `*.tgp` file.
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

abstract property
  schema object 128
AbsURL
  component 255
AcroForms
  processing PDF forms 166
actions
  compared to transformers 290
  defining 290
  input and output 289
  properties of 290, 392
  side effects 289
AddEmptyTagsTransformer
  component 255
AddEventAction
  component 291
AddHeaderModifier
  component 384
AdditionalInputPort
  component 153
AdditionalOutputPort
  component 155
AddString
  component 256
AddStringModifier
  component 385
advanced properties
  description 143
AFPToXML
  component 162
AggregateValues
  component 292
AllStructure
  component 244
AllStructureLocal
  component 245
alternative parsers
  selecting 214
AlternativeMappings
  component 347
Alternatives
  component 213
AlternativeSerializers
  component 332
AlternativeValidators
  component 393
anchor, repeating group
  highlighting all iterations 146
anchors
  defining 203
  extent of complex 212
  location in IntelliScript 203
  Marker and Content 201
  phase 206
anchors *(continued)*
  properties of 205
  reference 212
  relation to delimiters 202
  relation to XML 202
  serialization 325, 328
  using transformers 252
AppendListItems
  component 294
AppendValues
  component 295
arithmetic
  computations 296
ASN.1
  AsnToXml processor 53
  library 47
ASN.1 Streamer
  principle of operation 52
ASN.1Step
  editor 52
AsnToXml
  component 53
Assert
  Validation Rule element 413
assigning
  value to output 317
attribute properties
  schema object 131
attributeFormDefault
  schema object 132
attributes
  data holders 189
AttributeSearch
  component 240
autoInclude
  custom components 426

B

base property
  schema object 129
Base64Decoder
  component 256
Base64Encoder
  component 257
BCD
  convert 55
BCDToString
  component 55
BidiConvert
  component 257
BinaryFormat
  component 178
BIRT
  XmlToDocument report generator 168, 169

BIRT *(continued)*
XmlToDocument_372 report generator 168
XmlToDocument_45 report generator 169
block property
schema object 129
buffer input port
Data Processor transformation 27
buffer output port
Data Processor transformation 28

C

C/C++
custom components 424
CalculateValue
component 296
CDATADecode
component 258
CDATAEncode
component 258
ChangeCase
component 259
characters
special in IntelliScript 83
ChoiceStructure
component 245
ChoiceStructureLocal
component 246
choose hierarchy
Data Processor transformation 71, 76
COBOL
importing data definition 61
supported features 62
testing Parser 62
testing Serializer 63
code pages
transforming 264
XSD schema 190
collapse whitespace property
schema object 129
colors
highlighting in example source document 145
combinations
of lists 298
CombineValues
component 298
CommaDelimited
component 184
command-line interface
CM_console command 137
complex segments
streamer 369
complex type
advanced properties 131
ComplexSegment
component 376
ComplexXmlSegment
component 376
component properties
values, description 144
Component view
Data Processor transformation 26
component, global
defining 142
component, local
defining 142
component, Script
custom Java, developing 422
description 141
component, startup in Script
description 144
components
in IntelliScript 81
components, custom
C/C++ 424
components, custom Script
description 421
properties 422
components, Script
description 141
names 142
components,Script
properties, description 143
concatenation
strings 294
condition
ensuring in source document 305
Connect
component 247
Content
component 215
content anchors
highlighting in example source document 145
ContentSerializer
component 328, 333
CreateGuid
component 260
CreateList
component 299
CreateUUID
component 260
custom component
example 421
Java, developing 422
custom components
C/C++ 424
custom Script components
description 421
properties 422
CustomFormat
component 179
customize view
XMap Schema panel 89
CustomLog
component 300
cutting and pasting
Script components 146

D

data
validating 392
data holders
destroying occurrences 200
identifying source and target 357
indexing multiple-occurrence 352
mixed content 192
single or multiple occurrence 199
Data Process Transformation Wizard
description 47
Data Processing transformation
startup component 29

Data Processior transformation
denormalized relational output 79
Data Processor Events view
Data Processor transformation 37
viewing event log 39
Data Processor functions
description 111
Data Processor Hex Source view
description 26
Data Processor transformation
encoding settings 30
creating 40
denormalized relational input 75
description 25, 26
exporting as a service 44
input ports 27
mapping XML to ports 76
mapping XML to relational ports 71
normalized relational input 74
normalized relational output 78
output control settings 33
output ports 28
pivoted input 74
pivoted output 78
ports 27
processing settings 34
relational input
Data Processor transformation 72
relational output
Data Processor transformation 77
service parameter ports 28
settings 30
testing a library 123
testing in the Data Viewer 44
user logs 39
views 26
XMap settings 35
XML settings 35
Data Transformation service
import multiple services 45
importing to the Model repository 45
running from the command line 137
data types
searching for 211
DateAddICU
component 301
DateDiff
component 302
DateDiffICU
component 302
DateFormat
component 261
DateFormatICU
component 260
dates
format of 260
decimal
convert 55, 56
Default
component 99
Default statement
XMap editor 90
default transformers
in format 253
DelimitedSections
component 218
DelimitedSectionsSerializer
component 334
Delimiter
component 184
DelimiterHierarchy
component 185
delimiters
custom hierarchy 179
relation to anchors 202
denormalized relational input
Data Processor transformation 75
denormalized relational output
Data Processor transformation 79
derived data types
XSI type 192
design-time event log
description and location 38
direction property
of anchors 205
disabled property
selecting on menu 86
DocList
component 158
document processors
custom C++ 165
custom Java 165
defining 161
reference 162
running multiple 168
document, example source
description 145
DoNothingModifier
component 386
Dos96HebToAscii
component 263
DownloadFileToDataHolder
component 303
dp:as_XML
Data Processor function 111
dp:get_id
Data Processor function 111
dp:lookup
Data Processor function 111
dp:output
Data Processor function 111
DumpValues
component 304
dynamic
offset 241
search 243
DynamicTable
component 263

E

EbcdicToAscii
component 263
EDI
delimiters for parsing 185
editor, IntelliScript
description 146
editors
IntelliScript 80
elementFormDefault
schema object 132
elements
data holders 189
EmbeddedMapper
component 348

EmbeddedParser
component 221
EmbeddedSerializer
component 336
EmbeddedStructure
component 247
enclosed
group 222
EnclosedGroup
component 222
EnclosingDelimiters
component 185
EncodeAsUrl
component 264
Encoder
component 264
encoding
code page transformer 264
XSD schema 190
encoding guidelines
Data Processor transformation 33
encoding settings
Data Processor transformation 30
EnsureCondition
component 305
entering characters
non-keyboard 144
entry point, Script
description 144
Enumeration
component 395
enumeration property
schema object 128
errors
failure handling 389
validation failure handling 147
event log
custom events 291
viewing 39
event log, design-time
description and location 38
event types
Data Processor transformation 37
events
Data Processor transformation 37
handling 388
Events view
Data Processor transformation 37
Events view, Data Processor
viewing event log 39
example input file
Data Processor transformation 27
example source
description 145
setting 145
viewing 146
XMap Schema panel 89
Example Source
creating in Data Processor transformation 43
example source document, highlighting
description 145
example_source property
Mapper 345
Serializer 330
Excel
generating from XML 170
parsing as XML 163
ExcelToDataXml
component 163
ExcelToXml
component 163
ExcludeItems
component 308
ExpandFrameSet
component 165
ExternalJavaPreProcessor
component 165
ExternalPreProcessor
component 165
ExternalTransformer
component 265
extracting content
Content anchor 215
ExtractRecord
component 224

F

failure
effect on parent 389
failure event
Data Processor transformation 37
failure handling
variables for 196
failures
handing 389
handling 388
fatal error event
Data Processor transformation 37
file input port
Data Processor transformation 27
file ouptput port
Data Processor transformation 28
FileSearch
component 158
FindReplaceAnchor
component 225
fixed value property
schema object 128
footer segment
Streamer 369
format
preprocessors 188
FormatNumber
component 266
forms
processing PDF 166
frameset
parsing HTML 165
FromFloat
component 267
FromInteger
component 267
FromPackDecimal
component 268
FromSignedDecimal
component 268

G

generated prefix
changing for namespace 126

global component
defining 142
global components
defining 84
grid
XMap 90
group
performing actions on 227
repeating 232
Group
component 227
Group statements
XMap 93
GroupMapping
component 349
GroupSerializer
component 337

H

handling
failures 389
validation failures 147
header segment
Streamer 369
Hebrew
code-page conversion 269
hebrewBidi
component 269
HebrewDosToWindows
component 269
HebrewEBCDICOldCodeToWindows
component 269
hebUniToAscii
component 269
hebUtf8ToAscii
component 269
Hex Source view, Data Processor
description 26
Hexadecimal Source view
description 146
hexidecimal
convert 55, 56
HexSequenceToDecimal
component 55
HexToDecimal
component 56
hidden properties
showing 143
highlighting in example source document
description 145
HIPAA
validation 165
HIPAAValidator
component 165
HL7
component 186
HTML
removing tags 277
transforming entities 270
HtmlEntitiesToASCII
component 270
HtmlFormat
component 180
HtmlProcessor
component 188, 270

I

icons
IntelliScript 85
identifiers
IntelliScript 84
indexing
example 354
multiple-occurrence data holders 199
inherit by property
schema object 130
inherit from property
schema object 130
initialization
variables 198
InjectFP
component 270
InjectString
component 271
InlineTable
component 271
Input Expression Editor
Data Processor transformation 110
InputPort
component 159
Intelli mode
description 146
IntelliScript
defining anchors in 204
editing 82
icons used in 85
naming restrictions 84
IntelliScript editor
components and properties 81
description 146
IntelliScript Editor 80
IntelliScript Help view
description 146
invalid data
detecting 388
iterations
RepeatingGroup anchor 232

J

Java
custom component, developing 422
JavaScript
extensions 306
syntax reference 305
JavaTransformer
component 272

K

key
properties of 363
Key
component 363

L

LearnByExample
component 241

LengthEquals
  component 396
Library
  creating in Data Processor transformation 42
  element properties 121
  Overview 120
List
  Validation Rule element 414
list data types
  attributes 199
list types
  mapping to 211
lists
  combining 298
  creating 299
  multiple-occurrence data holders 199
  of variables 199
  sorting 318
local component
  defining 142
LocalFile
  component 159
LocalFile example source
  description 145
locations
  marking in source document 230
Locator
  component 365
LocatorByKey
  component 366
LocatorByOccurrence
  component 367
locators
  properties of 363
log
  definition 38
log, design-time event
  description and location 38
logs
  writing to 388, 408
Lookup
  Validation Rule element 414
LookupTransformer
  component 272
loop
  RepeatingGroup anchor 232

M

Map
  component 91, 308
mapper
  input validation 194
Mapper
  calling secondary 348
  component 345
  creating 342
  description 25
Mapper anchors
  properties of 344
  reference 346
mappers
  using indexing 354
Mappers
  properties of 344
  running in Parser 311
mapping statements
  XMap editor 90
Mapping Statements
  Cut and Paste 106
Mapplet
  reference 30
  running secondary 312, 315
Marker
  component 230
marker anchors
  highlighting in example source document 145
markers
  in Streamers 371
MarkerStreamer
  component 377
marking property
  of anchors 205, 377
maximum length
  schema object 128
maximum occurs
  schema object 128
MaxLength
  component 397
MaxNumber
  component 398
member types
  schema object 129
messages
  warning notifications 408
minimum length
  schema object 128
minimum occurs
  schema object 128
MinLength
  component 399
MinNumber
  component 400
missing data
  failure handling 389
missing text
  searching by optional Group 229
mixed content
  data holders in 192
  in schema 190
  mapping to 203
multiple occurrence
  data holders 199
  destroying occurrences 200
  variables 199
multiple-occurrence data holders
  combining 298
  creating lists in 299
  indexing 352
  mapping anchors to 203
  sorting 318

N

names
  IntelliScript 84
  Script components 142
namespace
  changing generated prefix 126
namespaces
  schema object 127
NewlineSearch
  component 241

nillible property
schema object 128
non-keyboard characters
entering 144
NormalizeClosingTags
component 274
normalized relational input
Data Processor transformation 74
normalized relational output
Data Processor transformation 78
Notification component 409
notification event
Data Processor transformation 37
notification events
StructureDefinition 239
NotificationGroup
component 409
NotificationHandler component 409
notifications
generating 388
triggering 310
writing messages 408
Notify
components 310
NotifyFailure component 410
NumberEquals
component 401
numbers
formatting 266

O

offset
dynamically defined 241
OffsetSearch
component 241
Option
component 98
Option statement
XMap editor 90
Option statements
XMap editor 96
optional failure
effect on parent 389
optional failure event
Data Processor transformation 37
optional property
failure handling 389
selecting on menu 86
setting 390
output control settings
Data Processor transformation 33
OutputDataHolder
component 323
OutputFile
component 323
OutputPort
component 159

P

packed decimals
numbers 268
parameters
passing to transformation 198
parser
Script components 149
Parser
component 150
description 25
for COBOL data 61
parsers
calling secondary 336
Parsers
calling secondary 221
running secondary 313
path
resolving relative 255
pattern matching
regular expressions 276
pattern property
schema object 128
patterns
segment opening and closing 369
PatternSearch
component 242
PDF
processing PDF forms 166
PDF conversion
configuring 173
PDF files
using PdfToTxt_4 processor 172
PDF output
XmlToDocument postprocessor 168, 169
XmlToDocument postprocessor_372 168
XmlToDocument postprocessor_45 169
PDF support
converting PDF files 166, 167
PdfFormToXml_1_00
component 166
PdfToTxt_3_02
component 166
PdfToTxt_4
component 167
using 172
performance issues
VarLastFailure variable 196
phase
of anchor search 206
phase property
of anchors 205
phases
nested 207
pivoted relational input
Data Processor transformation 74
pivoted relational output
Data Processor transformation 78
platform independence
Parsers 149
Ports view
Data Processor transformation 71, 76
Positional
component 186
postprocessor
XmlToDocument 168, 169
XmlToDocument_372 168
XmlToDocument_45 169
PostScript
component 187
PowerpointToTextML
component 167
pre-processors
defining 161

predicate statement
XPath expressions 107
preprocessors
document 161
format 188
ProcessByTransformers
component 168
ProcessorPipeline
component 168
processors
custom C++ 165
custom Java 165
document 161
reference 162
using transformers as 253
properties
custom Script components 422
in IntelliScript 81
of actions 290, 392
of anchors 205
of Mappers 344
of Serializers 330
of Streamers 375
Script components, description 143
Transformers 254
properties, advanced
description 143
properties, component
values, description 144
properties, hidden
showing 143
properties, simple
description 143
protocol specifications
ASN.1 schema 48

R

records
extracting and validating 236
extracting in StructureDefinition 224
RecordStructure
component 248
RecordStructureLocal
component 249
reference
anchors 212
delimiters 183
document processors 162
format preprocessors 188
formats 177
indexing 363
Mapper anchors 346
Mappers 345
parsers 150
serialization anchors 331
reference points
around anchors 205, 377
Marker anchor 230
of search scope 209
References view
Data Processor transformation 30
regex
regular expressions 276
regular expressions
syntax 276
RegularExpression
component 275
RemoveMarginSpace
component 277
RemoveRtfFormatting
component 277
RemoveTags
component 277
repeating group
highlighting in example source document 145
repeating group anchor
highlighting all iterations 146
Repeating Group statement
Run XMap statement
XMap editor 90
RunMapplet statement
XMap editor 90
XMap editor 90
Repeating Group statements
description 94
repeating segment
Streamer 369
RepeatingGroup
component 232
RepeatingGroupMapping
component 350
RepeatingGroupSerializer
component 338
Replace
component 278
replacing text
in source document 225
report generator
BIRT 168, 169
Repository view
Data Processor transformation 26
ResetVisitedPages
component 310
Resize
component 279
ResultFile
component 324
retrieving content
Content anchor 215
ReverseTransformer
component 279
right-to-left text
reversing 257, 269
rollback
after failure 389
Router statement
XMap editor 90
Router statements
XMap editor 96
RTF
component 187
RtfFormat
component 181
RtfProcessor
component 188, 280
RtfToASCII
component 280
RtfToTextML
component 168
Rule
Validation Rule element 415
Run XMap statement
mapping statement 100

run-time event log
Data Processor transformation 39
RunMapper
component 311
RunMapplet
component 312
RunMapplet statement
mapping statement 101
RunParser
component 313
RunPCWebService
component 315
RunSerializer
component 315
RunXMap
component 316

S

sample file
Avro 57
JSON 64
Parquet 67
XML 69
sample Script
importing 148
sample source file
defining in Data Processor transformation 41, 80
schema
ASN.1
protocol specification 48
ASN.1 protocol specification 48
Avro
schema 57
encoding 190
for COBOL data 61
JSON
schema 64
Parquet
definition 67
schema 67
XML
schema 69
schema files
adding to schema objects 126
editing 135
removing from schema objects 126
setting a default editor 134
schema object
elements advanced properties 129
abstract property 128
attribute properties 131
attributeFormDefault 132
block property 129
complex elements 130
complex elements advanced properties 131
editing a schema file 133
element properties 128
elementFormDefault 132
file location 132
importing 132
inherit by property 130
inherit from property 130
namespaces 127
overview 125
Overview view 125
schema files 126
schema object *(continued)*
setting a default editor 134
simple type 129
substitution group 129
synchronization 133
Schema object
Schema view 126
Schema view
Schema object 126
simple type advanced properties 130
schemas
Data Processor transformation 30
included schemas 190
IntelliScript representation 192
unsupported features 190
validation 193
XSD files 189
Script
creating in Data Processor transformation 41, 80
description 140
editor, IntelliScript 146
importing sample 148
samples 147
structure 141
Script component
custom Java, developing 422
description 141
Script components
description 141
names 142
properties, description 143
Script components, custom
description 421
properties 422
Script Help view
Data Processor transformation 26
Script mode
description 146
search
anchor direction 205
dynamically defined search string 243
search criteria
for anchors 207
search scope
adjusting 209
for anchors 207
searcher
components 210, 240
secondary Mapper
EmbeddedMapper anchor 348
secondary parser
EmbeddedParser anchor 336
secondary Parser
EmbeddedParser anchor 221
segments
processing in Streamer 369
SegmentSearch
component 242
select-and-click
defining anchors 204
SequenceStrructure
component 249
SequenceStructureLocal
component 250
serialization
mode 326
using transformers in 254

serialization anchors
properties of 330
reference 331
sequence of operation 329
serializer
input validation 194
Serializer
component 330
controlling auto-generation 325
for COBOL data 61
Serializers
properties of 330
running in Parser 315
troubleshooting auto-generated 327
service name
variable storing 196
service parameter port
Data Processor transformation 27
service parameter ports
Data Processor transformation 28
service parameters
passing to transformation 198
service, Data Transformation
running from the command line 137
ServiceLocation
variable 195
Settings view
Data Processor transformation 30
SetValue
component 317
SGML
component 187
signed decimals
numbers 268
simple properties
description 143
simple type
schema object 130
SImple Type view
schema object 129
SimpleSegment
component 379
SimpleXmlSegment
component 380
single occurrence
data holders 199
Sort
component 318
sorting
multiple-occurrence data holders 318
source
property 357, 358
source document, highlighting
description 145
SpaceDelimited
component 187
special characters
entering in IntelliScript 83
splitting
files 321
splitting large inputs
Streamer 368
startup component
Data Processor transformation 29
startup component, Script
description 144
streamer
complex segments 369
Streamer
component 381
creating 371
description 25
footer segment 369
header concatenation 370
header segment 369
JsonStreamer 377
output 370
repeating segment 369
segment opening and closing patterns 369
splitting large inputs 368
Streamers
properties of 375
StreamerVariable
component 382
string
convert 55
strings
concatenating 294, 295
StringSerializer
component 328
structured parsing
StructureDefinition 236
StructureDefinition
component 236
extracting records 224
substitution group
schema object 129
SubString
component 280
Swap
component 56
synchronize with editor
Data Processor transformation 44
system
variables 195
system time
variable 195

T

TabDelimited
component 187
table configuration editor
PdfToTxt_4 172
tables
processing PDF 172
target
property 357, 361
test a library
Data Processor transformation 123
test expression
Input XPath Expression Editor 110
Text
component 159
Text example source
description 145
text Streamer
principle of operation 369
TextFormat
component 181
TextML
XML schema 171
TextSearch
component 243

TGP file
for custom components 426
time
system 195
times
format of 260
ToFloat
component 281
ToInteger
component 282
ToPackDecimal
component 282
Trace
Validation Rule element 415
TransformationStartTime
component 283
TransformByParser
component 284
TransformByProcessor
component 285
TransformByService
component 285
Transformer
description 25
TransformerPipeline
component 286
transformers
as document preprocessors 253
compared to actions 290
custom DLL 265
custom Java 272
default 253
defining 252
in serialization 254
sequences of 253
using as document processors 168
using in anchors 252
Transformers
properties of 254
types
XSI 192
TypeSearch
component 244

U

UNIX
designing Parsers for 149
URL
relative to absolute 255
URL example source
description 145
user log
Data Processor transformation 39
variable defining location 196

V

ValidateByExpression
component 402
ValidateByPattern
component 403
ValidateByTransformer
component 404
ValidateByType
component 405
ValidateDate component 406
ValidateValue
component 319
validation
XML 193
XML input 194
XML Parser output 193
validation failures
handing 147
Validation Rule
Assert element 413
element 413–415
List element 414
Lookup element 414
Rule element 415
Trace element 415
Variable element 415
XPath editor 416
Validation Rule elements
XPath editor 416
Validation Rule Language
VRL 319
Validation Rule object
creating in Data Processor transformation 43
Validation Rules
definition 412
edit in external editor 419
editor 412, 419
importing Data Transformation service 419
ValidatorPipeline
component 407
validators
input data 392
values, component properties
description 144
VarCurrentPost
variable 195
VarCurrentURL
variable 195
Variable
component 198
Validation Rule element 415
variables
data holders 189
in Streamers 371
initialization 198
lists 199
mapping anchors to 197, 203
system 195
user-defined 195
using in actions 197
XMap editor 113
variety property
schema object 129
VarLastFailure
variable 196
VarLinkURL
variable 195
VarRequestedURL
variable 195
VarServiceInfo
variable 196
VarSystem
system time 195
view, Hexadecimal Source
description 146
view, IntelliScript Help
description 146

VRL
Validation Rule Language 319

W

warning event
Data Processor transformation 37
WellFormedModifier
component 386
Word
parsing as XML 168
WordToXml
component 168
WriteSegment
component 387
WriteValue
component 320

X

Xerces
XML validation 194
XMap
description 88
Group statements 93
Router statements 96
running in Mapper 316
running in Parser 316
running in Serializer 316
Schema panel 89
statement types 90
variables 113
XMap editor
grid 90
XMap object
creating in Data Processor transformation 42
XML
adding empty tags 255
mapping anchors to 202
schemas 189
validation 194
XSLT transformation 287
XML attributes
data holders 189
XML elements
data holders 189
XML Streamer
principle of operation 373
XmlFormat
component 182
XMLLookupTable
component 287
XmlSegment
component 382
XmlStreamer
component 52, 383
XmlToDocument
component 168, 169
XmlToDocument_372
component 168
XmlToDocument_45
component 169
XmlToExcel
component 170
XmlToXlsx
component 170
XPath
modified notation 192
XPath editor
Validation Rule elements 416
XPath Expression Editor
Data Processor transformation 110
XPath expressions
XMap 106
XSD
editors 189
schema encoding 190
unsupported schema features 190
XSD files
schemas 189
XSI types
mapping data holders 192
XSLT
running transformations 322
XSLTMap
component 322
XSLTTransformer
component 287