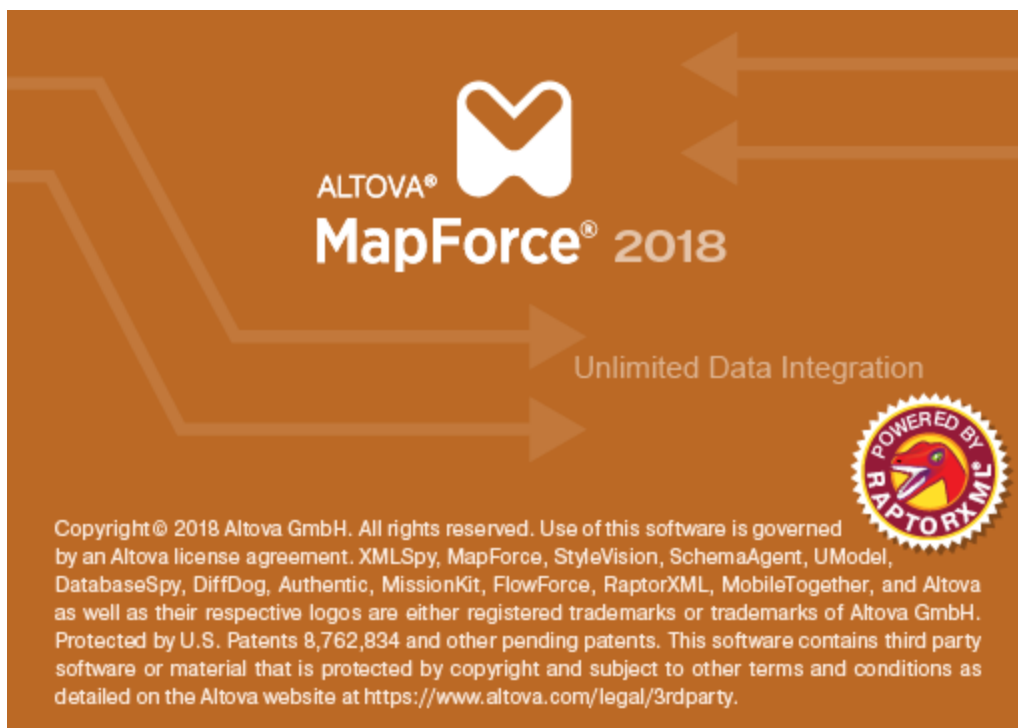


## User and Reference Manual

The image shows the cover of the Altova MapForce 2018 User and Reference Manual. The background is a solid orange color. In the center, the Altova logo (a white stylized 'M' inside a square) is positioned above the text 'ALTOVA®' and 'MapForce® 2018'. To the right of the logo, there are three horizontal arrows pointing left. Below the logo, there are two horizontal arrows pointing right. The text 'Unlimited Data Integration' is centered below the arrows. In the bottom right corner, there is a circular seal with a red border and the text 'POWERED BY RAPTORXML' around a central graphic of a red raptor head. At the bottom left, there is a paragraph of copyright and license information.

Copyright© 2018 Altova GmbH. All rights reserved. Use of this software is governed by an Altova license agreement. XMLSpy, MapForce, StyleVision, SchemaAgent, UModel, DatabaseSpy, DiffDog, Authentic, MissionKit, FlowForce, RaptorXML, MobileTogether, and Altova as well as their respective logos are either registered trademarks or trademarks of Altova GmbH. Protected by U.S. Patents 8,762,834 and other pending patents. This software contains third party software or material that is protected by copyright and subject to other terms and conditions as detailed on the Altova website at <https://www.altova.com/legal/3rdparty>.



# **Altova MapForce 2018 Basic Edition User & Reference Manual**

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Published: 2018

© 2018 Altova GmbH



---

# Table of Contents

<b>1</b>	<b>Altova MapForce 2018 Basic Edition</b>	<b>3</b>
1.1	What's new...	4
<b>2</b>	<b>Introduction</b>	<b>10</b>
2.1	Support Notes	11
2.2	What Is MapForce?	12
2.3	Basic Concepts	18
2.4	User Interface Overview	20
2.5	Conventions	26
<b>3</b>	<b>Tutorials</b>	<b>28</b>
3.1	Convert XML to New Schema	29
3.2	Map Multiple Sources to One Target	39
3.3	Work with Multiple Target Schemas	45
3.4	Process and Generate Files Dynamically	54
<b>4</b>	<b>Common Tasks</b>	<b>64</b>
4.1	Working with Mappings	65
4.1.1	Adding Components to the Mapping	65
4.1.2	Adding Components from a URL	66
4.1.3	Selecting a Transformation Language	69
4.1.4	Validating Mappings	70
4.1.5	Validating the Mapping Output	71
4.1.6	Previewing the Output	72
4.1.7	Text View Features	73
4.1.8	Searching in Text View	78
4.1.9	Previewing the XSLT Code	82
4.1.10	Generating XSLT Code	82
4.1.11	Working with Multiple Mapping Windows	83
4.1.12	Changing the Mapping Settings	84







---

5.5.1	Adding Simple Input Components .....	149
5.5.2	Simple Input Component Settings .....	150
5.5.3	Creating a Default Input Value .....	151
5.5.4	Example: Using File Names as Mapping Parameters .....	152
5.6	Returning String Values from a Mapping .....	155
5.6.1	Adding Simple Output Components .....	156
5.6.2	Example: Previewing Function Output .....	156
5.7	Using Variables .....	159
5.7.1	Adding Variables .....	160
5.7.2	Changing the Context and Scope of Variables .....	163
5.7.3	Example: Grouping and Subgrouping Records .....	165
5.8	Sorting Data .....	168
5.8.1	Sorting by Multiple Keys .....	170
5.8.2	Sorting with Variables .....	171
5.9	Filters and Conditions .....	174
5.9.1	Example: Filtering Nodes .....	175
5.9.2	Example: Returning a Value Conditionally .....	177
5.10	Using Value-Maps .....	180
5.10.1	Passing data through a Value-Map unchanged .....	183
5.10.2	Value-Map component properties .....	185
5.11	Mapping Node Names .....	188
5.11.1	Getting Access to Node Names .....	189
5.11.2	Accessing Nodes of Specific Type .....	196
5.11.3	Example: Map Element Names to Attribute Values .....	201
5.12	Mapping Rules and Strategies .....	205
5.12.1	Changing the Processing Order of Mapping Components .....	209
5.12.2	Priority Context node/item .....	212
5.12.3	Overriding the Mapping Context .....	213
<b>6</b>	<b>Data Sources and Targets</b>	<b>220</b>
6.1	XML and XML schema .....	221
6.1.1	Generating an XML Schema .....	221
6.1.2	XML Component Settings .....	222
6.1.3	Using DTDs as "Schema" Components .....	226
6.1.4	Derived XML Schema Types .....	227
6.1.5	QNames .....	229
6.1.6	Nil Values / Nillable .....	229



6.1.7	Comments and Processing Instructions .....	231
6.1.8	CDATA Sections .....	233
6.1.9	Wildcards - xs:any / xs:anyAttribute .....	234
6.1.10	Merging Data from Multiple Schemas .....	238
6.1.11	Declaring Custom Namespaces .....	240
6.2	HL7 Version 3 .....	243

## **7 Functions 246**

7.1	How To... .....	247
7.1.1	Add a Built-in Function to the Mapping .....	247
7.1.2	Add a Constant to the Mapping .....	248
7.1.3	Search for a Function .....	249
7.1.4	View a Function's Type and Description .....	250
7.1.5	Add or Delete Function Arguments .....	251
7.2	User-Defined Functions .....	252
7.2.1	Function parameters .....	257
7.2.2	Inline and regular user-defined functions .....	259
7.2.3	Creating a simple look-up function .....	261
7.2.4	User-defined function - example .....	265
7.2.5	Complex user-defined function - XML node as input .....	270
7.2.6	Complex user-defined function - XML node as output .....	276
7.2.7	Recursive user-defined mapping .....	280
7.3	Importing Custom XSLT 1.0 or 2.0 Functions .....	290
7.3.1	Example: Adding Custom XSLT Functions .....	291
7.3.2	Example: Summing Node Values .....	294
7.4	Regular Expressions .....	297
7.5	Function Library Reference .....	300
7.5.1	core   aggregate functions .....	300
7.5.2	core   conversion functions .....	305
7.5.3	core   file path functions .....	312
7.5.4	core   generator functions .....	314
7.5.5	core   logical functions .....	317
7.5.6	core   math functions .....	320
7.5.7	core   node functions .....	323
7.5.8	core   QName functions .....	325
7.5.9	core   sequence functions .....	326
7.5.10	core   string functions .....	338



---

7.5.11	xpath2   accessors .....	346
7.5.12	xpath2   anyURI functions .....	347
7.5.13	xpath2   boolean functions .....	348
7.5.14	xpath2   constructors .....	348
7.5.15	xpath2   context functions .....	349
7.5.16	xpath2   durations, date and time functions .....	350
7.5.17	xpath2   node functions .....	352
7.5.18	xpath2   numeric functions .....	354
7.5.19	xpath2   string functions .....	354
7.5.20	xslt   xpath functions .....	356
7.5.21	xslt   xslt functions .....	359
<b>8</b>	<b>Automating Mappings and MapForce</b>	<b>364</b>
8.1	Automation with RaptorXML Server .....	365
8.2	MapForce Command Line Interface .....	366
<b>9</b>	<b>Customizing MapForce</b>	<b>370</b>
9.1	Changing the MapForce Options .....	371
9.2	Altova Global Resources .....	373
9.2.1	Creating Global Resources .....	373
9.2.2	The Global Resources XML File .....	374
9.2.3	Example: Run Mapping with Variable Input Files .....	375
9.2.4	Example: Generate Output to Variable Folders .....	377
9.3	Customizing Keyboard Shortcuts .....	379
9.4	Catalog Files .....	381
9.5	Network Proxy Settings .....	386
<b>10</b>	<b>Menu Reference</b>	<b>390</b>
10.1	File .....	391
10.2	Edit .....	394
10.3	Insert .....	395
10.4	Component .....	397
10.5	Connection .....	398
10.6	Function .....	399
10.7	Output .....	400



---

10.8	View .....	401
10.9	Tools .....	403
10.10	Window .....	404
10.11	Help Menu .....	405

## **11 Appendices 412**

11.1	Engine information .....	413
11.1.1	XSLT and XQuery Engine Information .....	413
11.1.2	XSLT and XPath/XQuery Functions .....	418
11.2	Technical Data .....	486
11.2.1	OS and Memory Requirements .....	486
11.2.2	Altova XML Validator .....	486
11.2.3	Altova XSLT and XQuery Engines .....	486
11.2.4	Unicode Support .....	487
11.2.5	Internet Usage .....	487
11.3	License Information .....	488
11.3.1	Electronic Software Distribution .....	488
11.3.2	Software Activation and License Metering .....	489
11.3.3	Intellectual Property Rights .....	490
11.3.4	Altova End User License Agreement .....	490

## **12 Glossary 492**

12.1	C .....	493
12.2	F .....	494
12.3	G .....	495
12.4	I .....	496
12.5	J .....	497
12.6	M .....	498
12.7	O .....	499
12.8	P .....	500
12.9	S .....	501
12.10	T .....	502

## **Index**



# **Chapter 1**

---

**Altova MapForce 2018 Basic Edition**

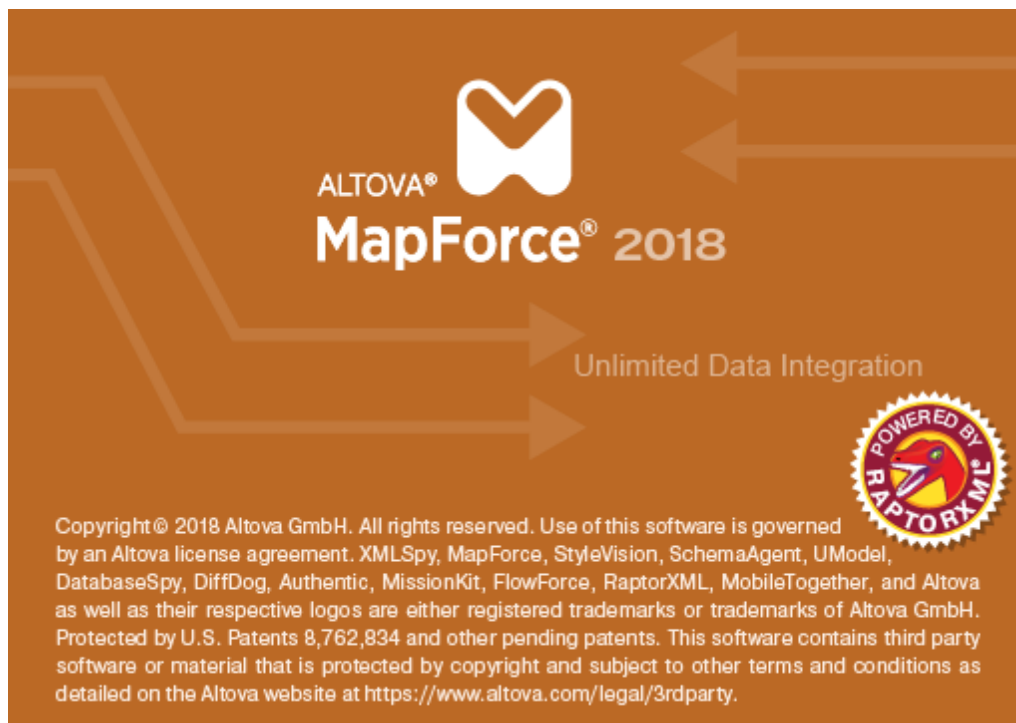






# 1 Altova MapForce 2018 Basic Edition

**MapForce® 2018 Basic Edition** is a visual data mapping tool for advanced data integration projects. MapForce® is a 32/64-bit Windows application that runs on Windows 7 SP1 with Platform Update, Windows 8, Windows 10, and Windows Server 2008 R2 SP1 with Platform Update or newer. 64-bit support is available for the Enterprise and Professional editions.



*Last updated: 21 June 2018*



## 1.1 What's new...

New in MapForce 2018 Release 2:

- Built-in functions, user-defined functions, and constants can be conveniently added to the mapping by double-clicking an empty area on the mapping (see [Add a Built-in Function to the Mapping](#) and [Add a Constant to the Mapping](#))
- Internal updates and optimizations

New in MapForce 2018:

- Internal updates and optimizations

New in MapForce 2017 Release 3:

- The text search options in the **Output** pane and the **XSLT** pane have been enhanced (see [Searching in Text View](#)). Also, text highlighting is available in the above-mentioned panes (see [Text Highlighting](#)).
- Internal updates and optimizations

New in MapForce 2017:

- It is possible to read node names from a source XML and map this information to a target. It is also possible to dynamically create new XML attributes or elements in a target based on values supplied from a source. See [Mapping Node Names](#).
- XML instance files can be created with custom namespaces, at element level (see [Declaring Custom Namespaces](#))
- Internal updates and optimizations

New in MapForce 2016 R2:

- More intuitive code folding in the [XSLT pane](#): collapsed text is displayed with an ellipsis symbol and can be previewed as a tooltip.
- You can search for all occurrences of a function within the active mapping (in the [Libraries window](#), right-click the function, and select **Find All Calls**).
- Internal updates and optimizations

New features in MapForce 2016:

- Improved generation of XSLT 1.0 code (generated stylesheets are easier to read and often faster to execute)
- Two new aggregate functions are available in the MapForce core library: [min-string](#) and [max-string](#). These functions enable you to get the minimum or maximum value from a sequence of strings.



New features in MapForce Version 2015 R4:

- Internal updates and optimizations

New features in MapForce Version 2015 R3 include:

- Option to [suppress](#) the `<?xml ... ?>` declaration in XML output
- New component type: [Simple Output](#)
- Internal updates and optimizations

New features in MapForce Version 2015 include:

- New `language` argument available in the [format-date](#) and [format-dateTime](#) functions
- New sequence function: [replicate-item](#)

New features in MapForce Version 2014 R2 include:

- New [sequence functions](#): generate sequence, item-at, etc.
- Ability to define [CDATA](#) sections in output components
- [Keeping](#) connections after deleting components
- Automatic highlighting of [mandatory items](#) in target components

New features in MapForce Version 2014 include:

- Integration of RaptorXML validator and basic support for [XML Schema 1.1](#)
- Integration of new RaptorXML XSLT engines
- [XML Schema Wildcard support](#), `xs:any` and `xs:anyAttribute`
- Support for [Comments and Processing Instructions](#) in XML target components

New features in MapForce Version 2013 R2 SP1 include:

- New super-fast transformation engine

New features in MapForce Version 2013 R2 include:

- Internal updates and optimizations.

New features in MapForce Version 2013 include:

- Internal updates and optimizations

New features in MapForce Version 2012 R2 include:

- New [Sort component](#) for XSLT 2.0, XQuery, and the Built-in execution engine
- User defined component names



New features in MapForce Version 2012 include:

- [Auto-alignment](#) of components in the mapping window
- Prompt to connect to [target parent](#) node
- Specific rules governing the [sequence](#) that components are processed in a mapping

New features in MapForce Version 2011R3 include:

- [Intermediate variables](#)

New features in MapForce Version 2011R2 include:

- [Find function](#) capability in Library window
- [Reverse](#) mapping
- Extendable [IF-ELSE](#) function
- [Node Name](#) and parsing functions in Core Library

New features in MapForce Version 2011 include:

- Ability to preview intermediate components in a [mapping chain](#) of two or more components connected to a target component (pass-through preview).
- Formatting functions for [dateTime](#) and [numbers](#) for all supported languages
- Enhancement to [auto-number](#) function

New features in MapForce Version 2010 Release 3 include:

- Support for [Nillable values](#), and xsi:nil attribute in XML instance files
- Ability to disable automatic [casting to target](#) types in XML documents

New features in MapForce Version 2010 Release 2 include:

- Automatic connection of identical [child connections](#) when moving a parent connection
- Ability to [tokenize input](#) strings for further processing

New features in MapForce Version 2010 include:

- [Multiple input/output](#) files per component
- Upgraded [relative path](#) support
- xsi:type support allowing use of [derived types](#)
- New internal data type system
- Improved user-defined [function navigation](#)
- Enhanced handling of [mixed content](#) in XML elements

New features in MapForce Version 2009 SP1 include:



- [Parameter order](#) in user-defined functions can be user-defined
- Ability to process XML files that are [not valid](#) against XML Schema
- [Regular](#) (Standard) user-defined functions now support complex hierarchical parameters

New features in MapForce Version 2009 include:

- EDI [HL7 versions 3.x](#) XML as source and target components
- [Grouping of nodes](#) or node content
- Ability to filter data based on a [nodes position](#) in a sequence
- [QName](#) support
- Item/node [search](#) in components

New features in MapForce Version 2008 Release 2 include:

- Ability to automatically [generate XML Schemas](#) for XML files
- Support for Altova [Global Resources](#)
- Performance optimizations

New features in MapForce Version 2008 include:

- [Aggregate](#) functions
- [Value-Map](#) lookup component
- Enhanced XML output options: [pretty print](#) XML output, omit [XML schema](#) reference and [Encoding settings](#) for individual components
- Various internal updates







# Chapter 2

---

## Introduction



## 2 Introduction

This introduction includes an overview of the MapForce features and user interface, the basic concepts in MapForce, as well as the conventions used in this documentation.



## 2.1 Support Notes

MapForce® is a 32/64-bit Windows application that runs on the following operating systems:

- Windows 7 SP1 with Platform Update, Windows 8, Windows 10
- Windows Server 2008 R2 SP1 with Platform Update or newer

64-bit support is available for the Enterprise and Professional editions.

For other technical information, see [Technical Data](#).



## 2.2 What Is MapForce?

**Altova website:**  [Data mapping tool](#)

MapForce is a Windows-based, multi-purpose IDE (integrated development environment) that enables you to transform data from one format to another, or from one schema to another, by means of a visual, "drag-and-drop" -style graphical user interface that does not require writing any program code. In fact, MapForce generates for you the program code which performs the actual data transformation (or data mapping). When you prefer not to generate program code, you can just run the transformation using the MapForce built-in transformation language (available in the MapForce Professional or Enterprise Editions).

Mappings designed with MapForce enable you to conveniently convert and transform data from and to a variety of file-based and other formats. Regardless of the technology you work with, MapForce determines automatically the structure of your data, or gives you the option to supply a schema for your data, or generate it automatically from a sample instance file. For example, if you have an XML instance file but no schema definition, MapForce can generate it for you, thus making the data inside the XML file available for mapping to other files or formats.

The technologies supported as mapping sources or targets are as follows.

MapForce Basic Edition	MapForce Professional Edition	MapForce Enterprise Edition
<ul style="list-style-type: none"> <li>• XML and XML schema</li> <li>• HL7 version 3.x (schema-based)</li> </ul>	<ul style="list-style-type: none"> <li>• XML and XML schema</li> <li>• Flat files, including comma-separated values (CSV) and fixed-length field (FLF) format</li> <li>• Databases (all major relational databases, including Microsoft Access and SQLite databases)</li> </ul>	<ul style="list-style-type: none"> <li>• XML and XML schema</li> <li>• Flat files, including comma-separated values (CSV) and fixed-length field (FLF) format</li> <li>• Data from legacy text files can be mapped and converted to other formats with MapForce FlexText</li> <li>• Databases (all major relational databases, including Microsoft Access and SQLite databases)</li> <li>• EDI family of formats (including UN/EDIFACT, ANSI X12, HL7, IATA PADIS, SAP IDoc, TRADACOMS)</li> <li>• JSON files</li> <li>• Microsoft Excel 2007 and later files</li> <li>• XBRL instance files and taxonomies</li> </ul>

Based on the MapForce edition, you can choose the preferred language for your data transformation as follows.

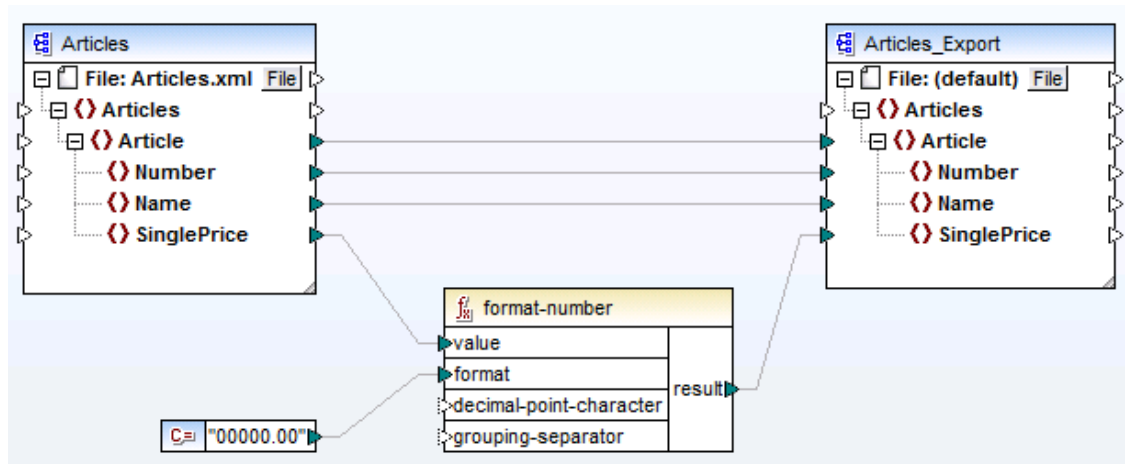


MapForce Basic Edition	MapForce Professional Edition	MapForce Enterprise Edition
<ul style="list-style-type: none"> <li>• XSLT 1.0</li> <li>• XSLT 2.0</li> </ul>	<ul style="list-style-type: none"> <li>• MapForce built-In transformation language</li> <li>• XSLT 1.0</li> <li>• XSLT 2.0</li> <li>• XQuery</li> <li>• Java</li> <li>• C#</li> <li>• C++</li> </ul>	<ul style="list-style-type: none"> <li>• MapForce built-In transformation language</li> <li>• XSLT 1.0</li> <li>• XSLT 2.0</li> <li>• XQuery</li> <li>• Java</li> <li>• C#</li> <li>• C++</li> </ul>

You can preview the result of all transformations, as well as the generated XSLT or XQuery code without leaving the graphical user interface. Note that, as you design or preview mappings, MapForce validates the integrity of your schemas or transformations and displays any validation errors in a dedicated window, so that you can immediately review and address them.

When you choose Java, C#, or C++ as transformation language, MapForce generates the required projects and solutions so that you can open them directly in Visual Studio or Eclipse, and run the generated data mapping program. For advanced data integration scenarios, you can also extend the generated program with your own code, using Altova libraries and the MapForce API.

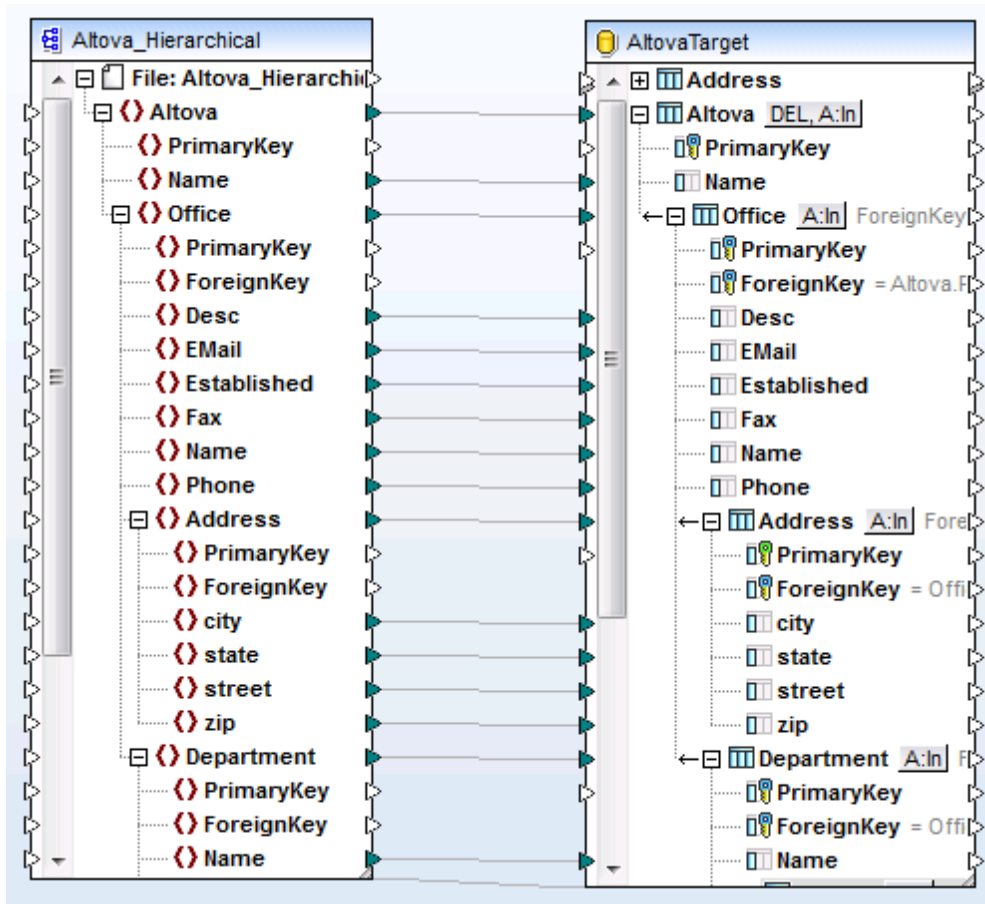
In MapForce, you design all mapping transformations visually. For example, in case of XML, you can connect any element, attribute, or comment in an XML file to an element or attribute of another XML file, thus instructing MapForce to read data from the source element (or attribute), and write it to the target element (or attribute).



Sample data transformation between two XML files

Likewise, when working with databases in MapForce Professional or Enterprise Editions, you can see any database column in the MapForce mapping area and map data to or from it by making visual connections. As with other Altova MissionKit products, when setting up a database connection from MapForce, you can flexibly choose the database driver and the connection type (ADO, ODBC, or JDBC) according to your existing infrastructure and data mapping needs. Additionally, you can visually build SQL queries, use stored procedures, or query a database directly (support varies by database type, edition and driver).





Sample data transformation between an XML file and a database

In a very simple scenario, a mapping design created with MapForce could be resumed as "read data from the source X and write it to target Y". However, you can easily design MapForce scenarios such as "read data from the source X and write it to target Y, and then read data from the source Y and write it to the target Z". These are known as "pass-through", or "chained" mappings, and enable you to access your data at an intermediary stage in the transformation process (in order to save it to a file, for example).

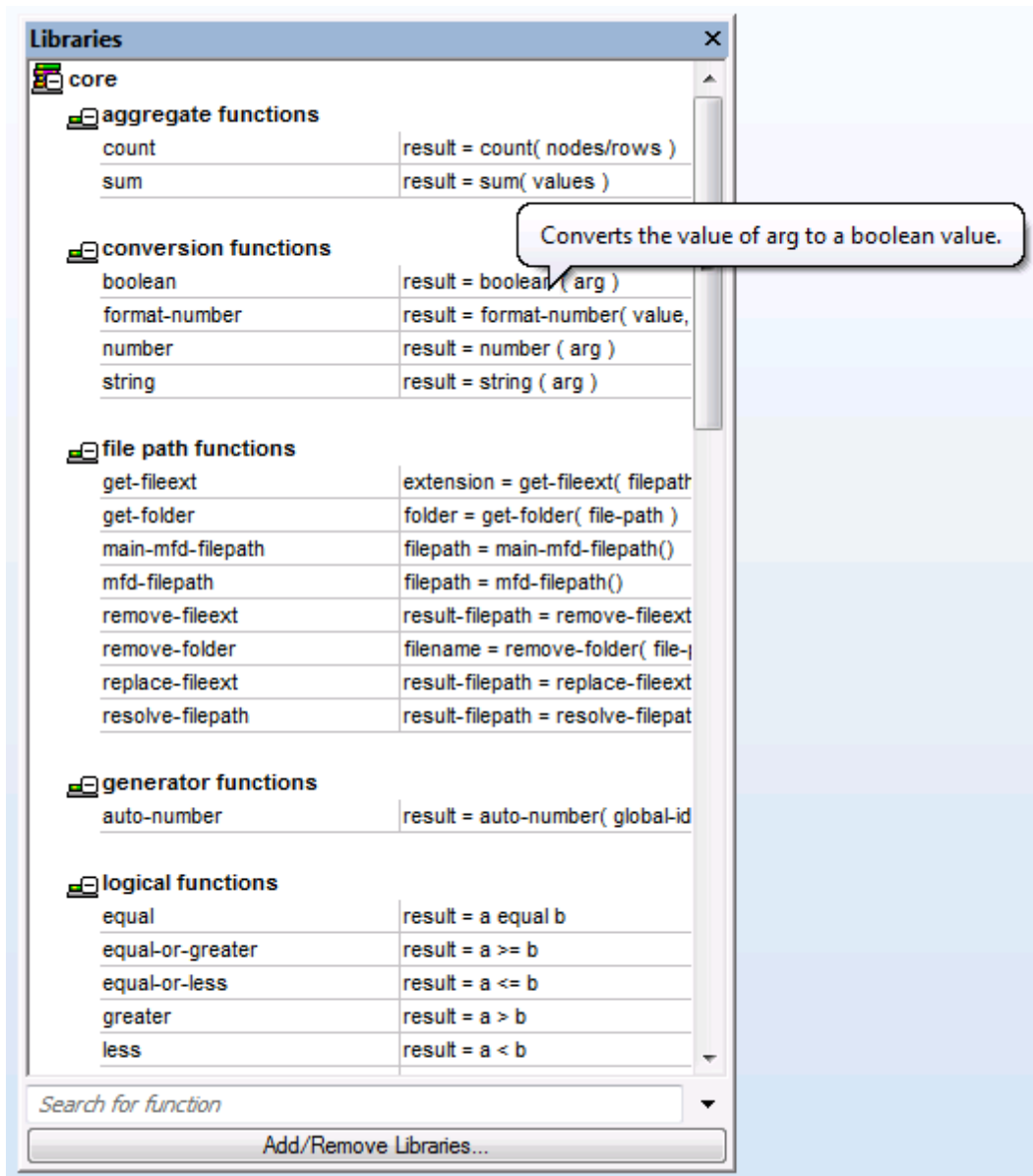
Note that the data mappings you can create in MapForce are not limited to single, predefined files. In the same transformation, you can process dynamically multiple input files from a directory and generate multiple output files. Therefore, you can have scenarios such as "read data from multiple X files and write it to a single Y file", or "read file X and generate multiple files Y", and so on.

Importantly, in the same transformation, you can mix multiple sources and multiple targets, which can be of any type supported by your MapForce edition. For example, in case of MapForce Professional or Enterprise, this makes it possible to merge data from two different databases into a single XML file. Or, you can merge data from multiple XML files, and write some of the data to one database, and some of the data to another database. You can preview the SQL statements before committing them to the database.



Direct conversion of data from a source to a target is not typically the only thing you want to achieve. In many cases, you might want to process your data in a particular way (for example, sort, group or filter it) before it reaches the destination. For this reason, MapForce includes, on one hand, miscellaneous functional components that are simplified programming language constructs (such as constants, variables, SQL-WHERE conditions, Filter and Sort components). On the other hand, MapForce includes rich and extensible function libraries which can assist you with virtually any kind of data manipulation.

If necessary, you can extend the built-in library either with functions you design in MapForce directly (the so-called User-Defined Functions, or UDF), or with functions or libraries created externally in XSLT, XQuery, Java, or C# languages.



Libraries pane (MapForce Basic Edition)



When your data mapping design files become too many, you can organize them into mapping projects (available in MapForce Professional and Enterprise edition). This allows for easier access and management. Importantly, you can generate program code from entire projects, in addition to generating code for individual mappings within the project.

For advanced data processing needs (such as when running mapping transformations with the MapForce Server API), you can design a mapping so that you can pass values to it at run-time, or get a simple string value from it at run-time. This feature also enables you to quickly test the output of functions or entire mappings that produce a simple string value. The Professional and Enterprise editions of MapForce also include components that enable you to perform run-time string parsing and serialization, similar to how this works in many other programming languages.

With MapForce Enterprise Edition, you can visually design SOAP 1.0 and SOAP 2.0 Web services based on Web Service Language Definition (WSDL) files. You can also call and get data from a WSDL 1.0 or a WSDL 2.0 Web service from within a mapping. This includes Web services available both through the HTTP and HTTPS protocols, as well as Web services which require that the caller uses the WS-Security mechanism, or HTTP authentication.

With MapForce Professional and Enterprise Editions, you can generate detailed documentation of your mapping design files, in HTML, Word 2007+, or RTF formats. Documentation design can be customized (for example, you can choose to include or exclude specific components from the documentation).

If you are using MapForce alongside other Altova MissionKit products, MapForce integrates with them as well as with the Altova server-based products, as shown in the following table.

MapForce Basic Edition	MapForce Professional Edition	MapForce Enterprise Edition
You can choose to run the generated XSLT directly in MapForce and preview the data transformation result immediately. When you need increased performance, you can process the mapping using RaptorXML Server, an ultra-fast XML transformation engine.		
If XMLSpy is installed on the same machine, you can conveniently open and edit any supported file types, by opening XMLSpy directly from the relevant MapForce contexts (for example, the <b>Component   Edit Schema Definition in XMLSpy</b> menu command is available when you click an XML component).		
	You can run data transformations either directly in MapForce, or deploy them to a different machine and even operating system for command-line or automated execution. More specifically, you can design mappings on Windows, and run them on a Windows, Linux, or Mac server machine which runs MapForce Server (either standalone or under FlowForce Server management).	
	If StyleVision is installed on the same machine, you can design or reuse existing StyleVision Power Stylesheets and preview the result of the mapping transformations as HTML, RTF, PDF, or Word 2007+ documents.	

MapForce Professional and Enterprise edition can be installed as a plug-in of Visual Studio and Eclipse integrated development environments. This way, you can design mappings and get



access to MapForce functionality without leaving your preferred development environment.

In MapForce, you can completely customize not only the look and feel of the development environment (graphical user interface), but also various other settings pertaining to each technology and to each mapping component type, for example:

- When mapping to or from XML, you can choose whether to include a schema reference, or whether the XML declaration must be suppressed in the output XML files. You can also choose the encoding of the generated files (for example, UTF-8).
- When mapping to or from databases, you can define settings such as the time-out period for executing database statements, whether MapForce should use database transactions, or whether it should strip the database schema name from table names when generating code.
- In case of XBRL, you can select the structure views MapForce should display (such as the "Presentation and definition linkbases" view, the "Table Linkbase" View, or the "All concepts" view).

All editions of MapForce are available as a 32-bit application. The MapForce Professional and Enterprise editions are additionally available as a 64-bit application.

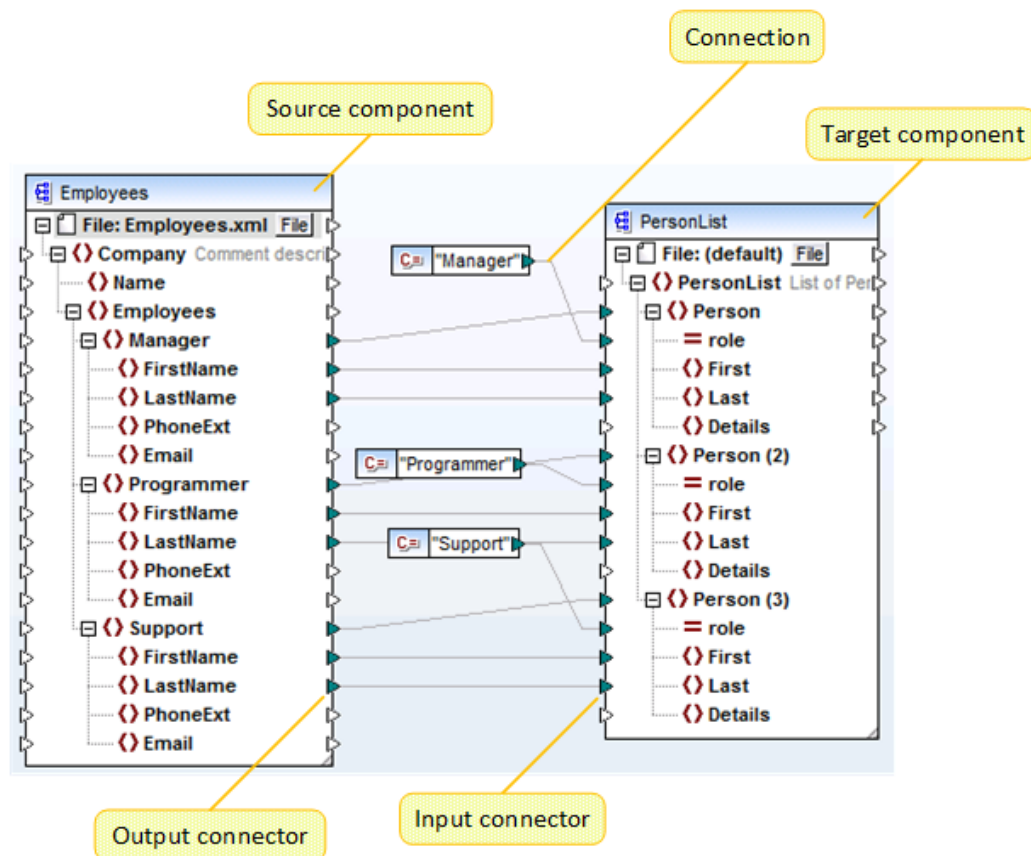


## 2.3 Basic Concepts

This section outlines the basic concepts that will help you get started with data mapping.

### Mapping

A MapForce mapping design (or simply "mapping") is the visual representation of how data is to be transformed from one format to another. A mapping consists of [components](#) that you add to the MapForce mapping area in order to create your data transformations (for example, convert XML documents from one schema to another). A valid mapping consists of one or several [source components](#) connected to one or several [target components](#). You can run a mapping and preview its result directly in MapForce. You can generate code and execute it externally. You can also compile a mapping to a MapForce execution file and automate mapping execution using MapForce Server or FlowForce Server. MapForce saves mappings as files with .mfd extension.



Basic structure of a MapForce mapping

### Component

In MapForce, the term "component" is what represents visually the structure (schema) of your data, or how data is to be transformed (functions). Components are the central building pieces of



any [mapping](#). On the mapping area, components appear as rectangles. The following are examples of MapForce components:

- Constants
- Filters
- Conditions
- Function components
- EDI documents (UN/EDIFACT, ANSI X12, HL7)
- Excel 2007+ files
- Simple [input components](#)
- Simple [output components](#)
- XML Schemas and DTDs

## Connector

A connector is a small triangle displayed on the left or right side of a [component](#). The connectors displayed on the left of a component provide data entry points *to that component*. The connectors displayed on the right of a component provide data exit points *from that component*.

## Connection

A connection is a line that you can draw between two [connectors](#). By drawing connections, you instruct MapForce to transform data in a specific way (for example, read data from an XML document and write it to another XML document).

## Source component

A source component is a [component](#) from which MapForce reads data. When you run the [mapping](#), MapForce reads the data supplied by the connector of the source component, converts it to the required type, and sends it to the connector of the [target component](#).




## Target component

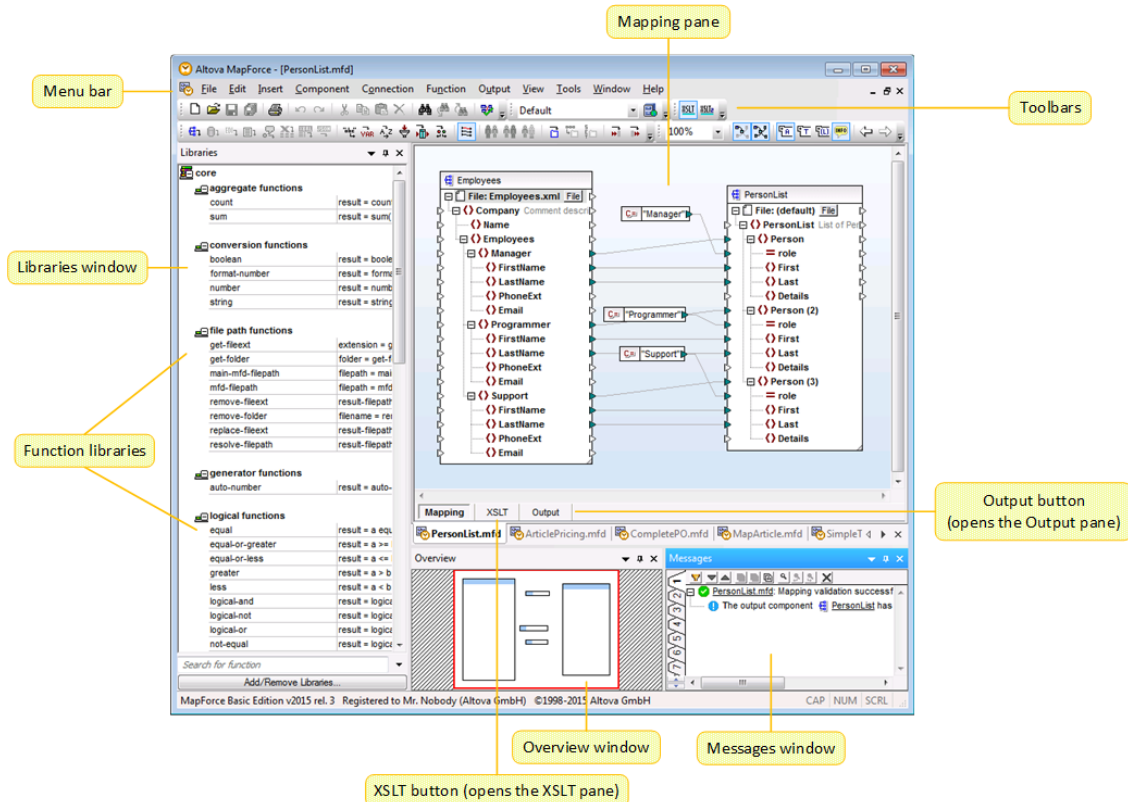
A target component is a [component](#) to which MapForce writes data. When you run the [mapping](#), a target component instructs MapForce to either generate a file (or multiple files) or output the result as a string value for further processing in an external program. A target component is the opposite of a [source component](#).



## 2.4 User Interface Overview

The graphical user interface of MapForce is organized as an integrated development environment. The main interface components are illustrated below. You can change the interface settings by using the menu command **Tools | Customize**.

Use the    buttons displayed in the upper-right corner of each window to show, hide, pin, or dock it. If you need to restore toolbars and windows to their default state, use the menu command **Tools | Restore Toolbars and Windows**.



*MapForce graphical user interface (MapForce Basic Edition)*

### Menu Bar and Toolbars

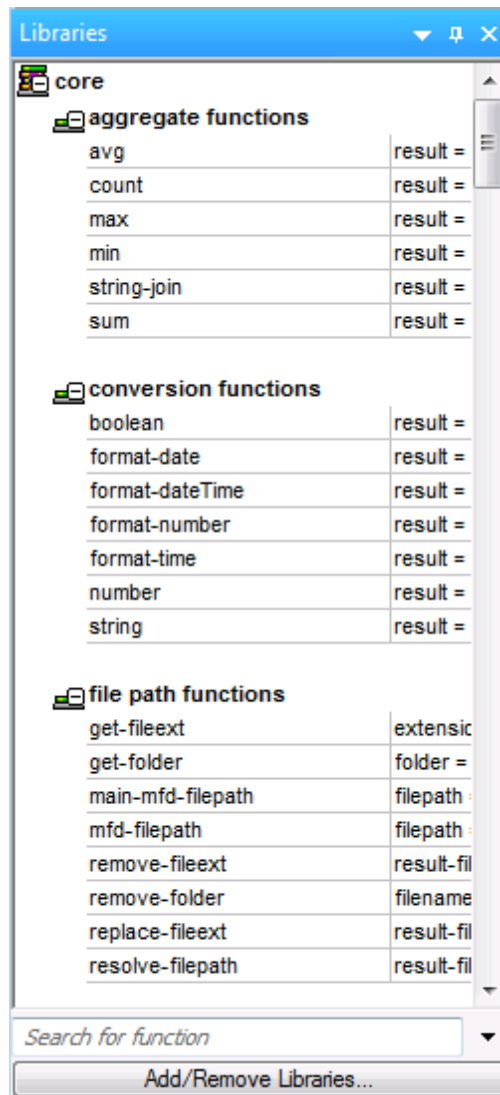
The Menu Bar displays the menu items. Each toolbar displays a group of buttons representing MapForce commands. You can reposition the toolbars by dragging their handles to the desired locations.

### Libraries window

The Libraries window lists the MapForce built-in functions, organized by library. The list of available functions changes based on the transformation language you select. If you have



created user-defined functions, or if you imported external libraries, they also appear in the Libraries window.



To search functions by name or by description, enter the search value in the text box at the bottom of the **Libraries** window. To find all occurrences of a function (within the currently active mapping), right-click the function, and select **Find All Calls** from the context menu. You can also view the function data type and description directly from the **Libraries** window. For more information, see [Working with Functions](#).

## Mapping pane

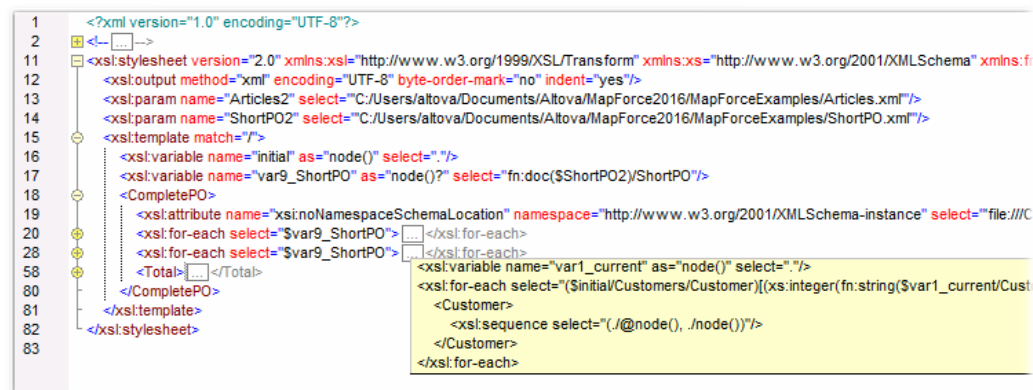
The Mapping pane is the working area where you design [mappings](#). You can add mapping components (such as files, schemas, constants, variables, and so on) to the mapping area from the **Insert** menu (see [Adding Components to the Mapping](#)). You can also drag into the Mapping pane functions displayed in the Libraries window (see [Working with Functions](#)).




## XSLT (XSLT2) pane

The XSLT (or XSLT2) pane displays the XSLT 1.0 (or 2.0) transformation code generated from your mapping. To switch to this pane, select XSLT (or XSLT 2) as transformation language, and then click the **XSLT** tab (or **XSLT2** tab, respectively).

This pane provides line numbering and code folding functionality. To expand or collapse portions of code, click the "+" and "-" icons at the left side of the window. Any portions of collapsed code are displayed with an ellipsis symbol. To preview the collapsed code without expanding it, move the mouse cursor over the ellipsis. This opens a tooltip that displays the code being previewed, as shown in the image below. Note that, if the previewed text is too big to fit in the tooltip, an additional ellipsis appears at the end of the tooltip.

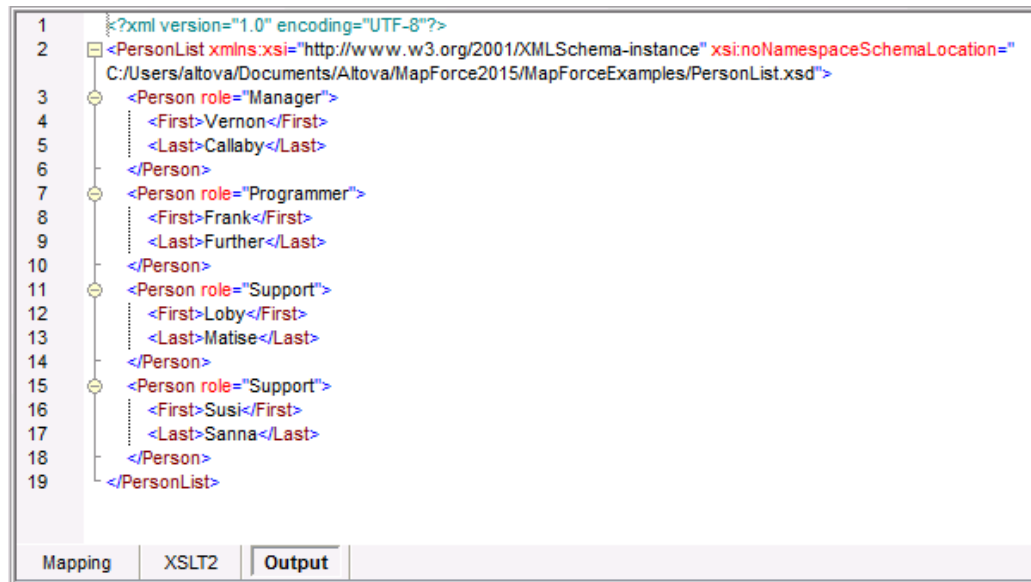


To configure the display settings (including indentation, end of line markers, and others), right-click the pane, and select **Text View Settings** from the context menu. Alternatively, click the **Text View Settings** (  ) toolbar button.

## Output pane

The Output pane displays the result of the mapping transformation (for example, an XML file), when you click the **Output** button. If the mapping generates multiple files, you can navigate sequentially through each generated file.

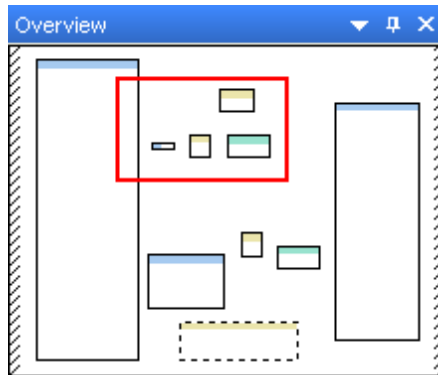




This pane also provides line numbering and code folding functionality, which works in a similar way as in the XSLT pane (see above).

## Overview window

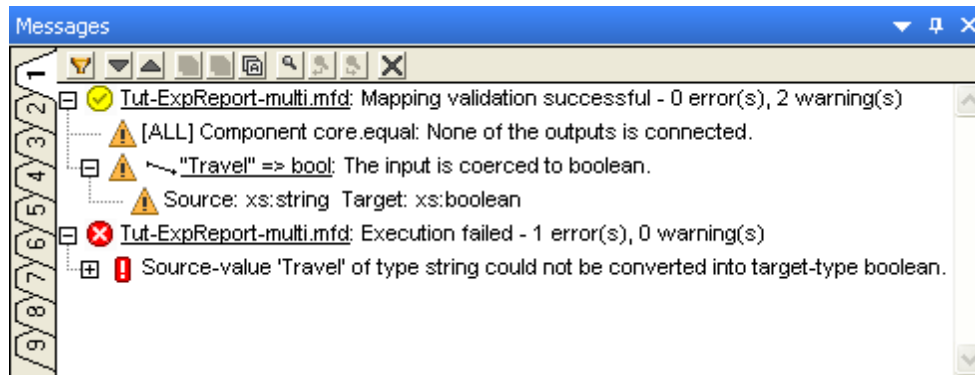
The Overview window gives a bird's-eye view of the Mapping pane. Use it to navigate quickly to a particular location on the mapping area when the size of the mapping is very large. To navigate to a particular location on the mapping, click and drag the red rectangle.



## Messages window

The Messages window shows messages, errors, and warnings when you execute a mapping (see [Previewing the Output](#)) or perform a mapping validation (see [Validating Mappings](#)).





To highlight on the mapping area the component or structure which triggered the information, warning, or error message, click the underlined text in the Messages window.

The results of a mapping execution or validation operation is displayed in the Messages window with one of the following status icons:

Icon	Description
	Operation completed successfully.
	Operation completed with warnings.
	Operation has failed.










The Message window may additionally display any of the following message types: information messages, warnings, and errors.

Icon	Description
	Denotes an information message. Information messages do not stop the mapping execution.
	Denotes a warning message. Warnings do not stop the mapping execution. They may be generated, for example, when you do not create connections to some mandatory input connectors. In such cases, output will still be generated for those component where valid connections exist.
	Denotes an error. When an error occurs, the mapping execution fails, and no output is generated. The preview of the XSLT or XQuery code is also not possible.

Other buttons in the Messages window enable you to take the following actions:

Icon	Description
	Filter messages by severity (information messages, errors, warnings). Select <b>Check All</b> to include all severity levels (this is the default behaviour).



Icon	Description
	Select <b>Uncheck All</b> to remove all severity levels from the filter. In this case, only the general execution or validation status message is displayed.
	Jump to next line.
	Jump to previous line.
	Copy the selected line to clipboard.
	Copy the selected line to clipboard, including any lines nested under it.
	Copy the full contents of the Messages window to clipboard.
	Find a specific text in the Messages window. Optionally, to find only words, select <b>Match whole word only</b> . To find text while preserving the upper or lower case, select <b>Match case</b> .
	Find a specific text starting from the currently selected line up to the end.
	Find a specific text starting from the currently selected line up to the beginning.
	Clear the Messages window.

When you work with multiple mapping files simultaneously, you might want to display information, warning, or error messages in individual tabs for each mapping. In this case, click the numbered tabs available on the left side of the Messages window before executing or validating the mapping.

### Application status bar

The application status bar appears at the bottom of the application window, and shows application-level information. The most useful of this information are the tooltips that are displayed here when you move the mouse over a toolbar button. If you are using the 64-bit version of MapForce, the application name appears in the status bar with the suffix (x64). There is no suffix for the 32-bit version.



## 2.5 Conventions

### Example files

Most of the data mapping design files (files with .mfd extension, as well as other accompanying instance files) illustrated or referenced in this documentation are available in the following folders:

- C:\Users\<username>\Documents\Altova\MapForce2018\MapForce Examples
- C:\Users\<username>\Documents\Altova\MapForce2018\MapForce Examples  
  \Tutorials

The example mappings and instance files accompanying MapForce illustrate most aspects of how it works, and you are highly encouraged to experiment with them as you learn about MapForce. When in doubt about the possible effects of making changes to the MapForce original examples, create back-ups before changing them.

### Graphical user interface

Some of the images (screen shots) accompanying this documentation depict graphical user interface elements that may not be applicable to your MapForce edition. In relevant contexts, images typically include the name of the source mapping design (\*.mfd) file, as well as the edition of MapForce in which the graphic was produced.



# Chapter 3

---

## Tutorials



## 3 Tutorials

The MapForce tutorials are intended to help you understand and use the basic data transformation capabilities of MapForce in a short amount of time. You can regard these tutorials as a "crash course" of MapForce. While the goal is not to illustrate completely all MapForce features, you will be guided through the MapForce basics step-by-step, so it is recommended that you follow the tutorials sequentially. It is important that you understand each concept before moving on to the next one, as the tutorials gradually grow in complexity. Basic knowledge of XML and XML schema will be advantageous.

### [Convert XML to New Schema](#)

This tutorial shows you how to convert data from an XML structure to another using the XSLT 2.0 language, without writing any code. You will also learn about MapForce sequences and items, creating mapping connections, using a function, validating and previewing a mapping, as well as saving the resulting output to the disk.

### [Map Multiple Sources to One Target](#)

This tutorial shows you how to read data from two XML files with different schema and merge it into a single target XML file. You will also learn how to change the name and instance files of each mapping component, and the concept of "duplicate inputs".

### [Work with Multiple Target Schemas](#)

This tutorial shows you how to work with more complex mappings that produce two or more target outputs. More specifically, you will learn how to generate, in the same mapping, an XML file that stores a list of book records, and another XML file that contains only a subset of the books in the first file, filtered by a specific publication year. To support filtering data, you will use a **Filter** component, a function and a numeric constant.

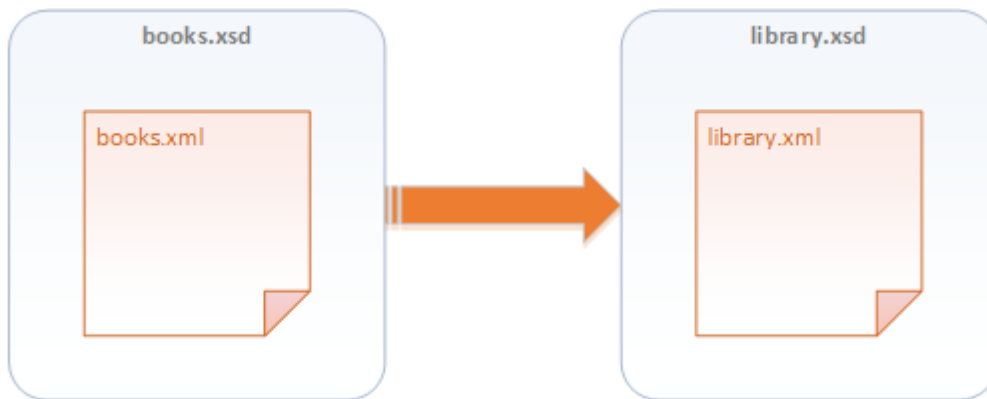
### [Process and Generate Files Dynamically](#)

This tutorial shows you how to read data from multiple XML instance files located in the same folder and write it to multiple XML files generated on the fly. You will also learn about stripping the XML and schema declarations and using functions to concatenate strings and extract file extensions.



## 3.1 Convert XML to New Schema

This tutorial shows you how to convert data between two XML files, while helping you learn the basics of the MapForce development environment. Both XML files store a list of books, but their elements are named and organized in a slightly different way (that is, the two files have different schemas).



*Abstract model of the data transformation*

The code listing below shows sample data from the file that will be used as data source (for the sake of simplicity, the XML and the namespace declarations are omitted).

```
<books>
  <book id="1">
    <author>Mark Twain</author>
    <title>The Adventures of Tom Sawyer</title>
    <category>Fiction</category>
    <year>1876</year>
  </book>
  <book id="2">
    <author>Franz Kafka</author>
    <title>The Metamorphosis</title>
    <category>Fiction</category>
    <year>1912</year>
  </book>
</books>
```

*books.xml*

This is how data should look in the target (destination) file:

```
<library>
  <last_updated>2015-06-02T16:26:55+02:00</last_updated>
  <publication>
    <id>1</id>
    <author>Mark Twain</author>
```



```
<title>The Adventures of Tom Sawyer</title>
<genre>Fiction</genre>
<publish_year>1876</publish_year>
</publication>
<publication>
  <id>2</id>
  <author>Franz Kafka</author>
  <title>The Metamorphosis</title>
  <genre>Fiction</genre>
  <publish_year>1912</publish_year>
</publication>
</library>
```

*library.xml*

As you may have noticed, some element names in the source and target XML are not the same. Our goal is to populate the <author>, <title>, <genre> and <publish\_year> elements of the target file from the equivalent elements in the source file (<author>, <title>, <category>, <year>). The attribute id in the source XML file must be mapped to the <id> element in the target XML file. Finally, we must populate the <last\_updated> element of the target XML file with the date and time when the file was last updated.

To achieve the required data transformation, let's take the following steps.

### Step 1: Select XSLT2 as transformation language



You can do this in one of the following ways:

- Click the **XSLT2** () toolbar button.
- On the **Output** menu, click **XSLT 2.0**.

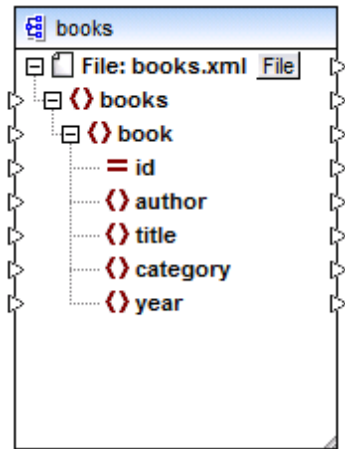
### Step 2: Add the source XML file to the mapping

The source XML file for this mapping is located at the following path: **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\books.xml**. You can add it to the mapping in one of the following ways:


- Click the **Insert XML Schema/File** () toolbar button.
- On the **Insert** menu, click **XML Schema/File**.
- Drag the XML file from Windows Explorer into the mapping area.




Now that the file has been added to the mapping area, you can see its structure at a glance. In MapForce, this structure is known as a mapping component, or simply [component](#). You can expand elements in the component either by clicking the collapse () and expand icons () , or by pressing the + and - keys on the numeric keypad.





Mapping component


To move the component inside the mapping pane, click the component header and drag the mouse to a new position. To resize the component, drag the corner of the component . You can also double-click the corner so that MapForce adjusts the size automatically.

The top level node  represents the file name; in this particular case, its title displays the name of the XML instance file. The XML elements in the structure are represented by the  icon, while XML attributes are represented by the  icon.

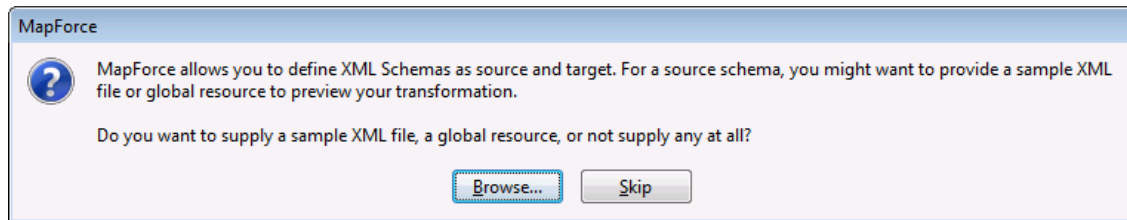
The small triangles displayed on both sides of the component represent data inputs (if they are on the left side) or outputs (when they are on the right side). In MapForce, they are called input connectors and output connectors, respectively.

### Step 3: Add the target XML schema to the mapping

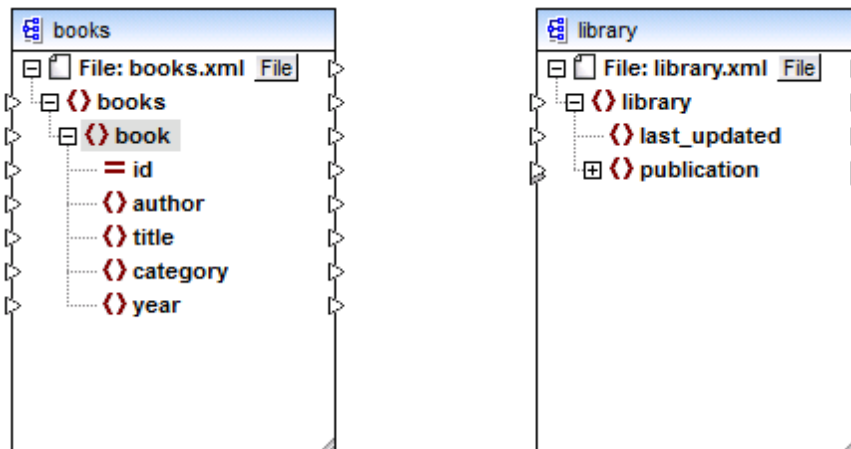
To generate the target XML, we will use an existing XML schema file. In a real-life scenario, this file may have been provided to you by a third party, or you can create it yourself with a tool such as XMLSpy. If you don't have a schema file for your XML data, MapForce prompts you to generate it whenever you add to the mapping an XML file without an accompanying schema or schema reference.

For this particular example, we are using an existing schema file available at: **<Documents> \Altova\MapForce2018\MapForceExamples\Tutorial\library.xsd**. To add it to the mapping, follow the same steps as with the source XML file (that is, click the **Insert XML Schema/File** () toolbar button). Click **Skip** when prompted by MapForce to supply an instance file.





At this stage, the mapping design looks as follows:

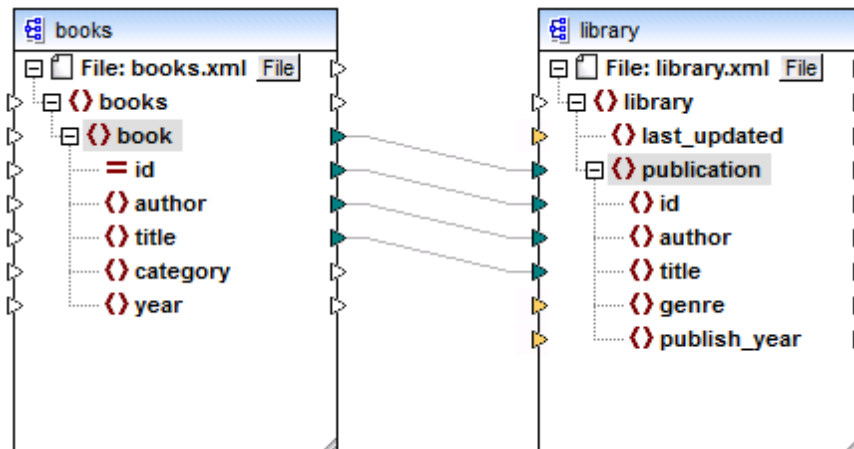


#### Step 4: Make the connections

For each `<book>` in the source XML file, we want to create a new `<publication>` in the target XML file. We will therefore create a mapping connection between the `<book>` element in the source component and the `<publication>` element in the target component. To create the mapping connection, click the output connector (the small triangle) to the right of the `<book>` element and drag it to the input connector of the `<publication>` element in the target.

When you do this, MapForce may automatically connect all elements which are children of `<book>` in the source file to elements having the same name in the target file; therefore, four connections are being created simultaneously. This behavior is called "Auto Connect Matching Children" and it can be disabled and customized if necessary.






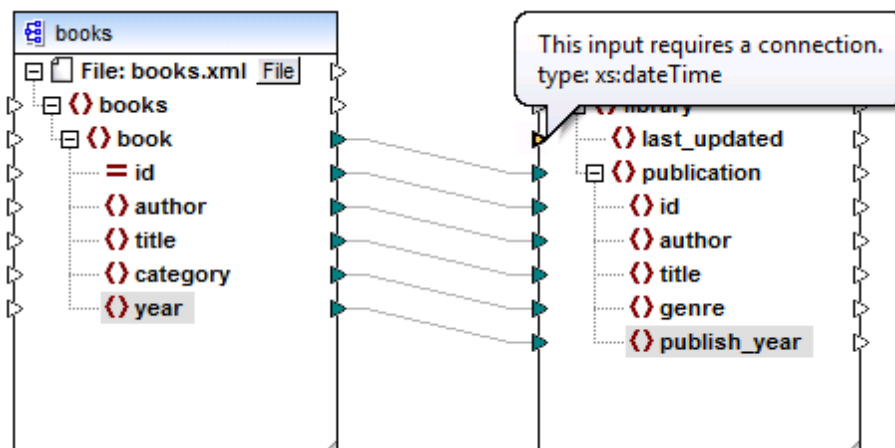
You can enable or disable the "Auto Connect Matching Children" behavior in one of the following ways:

- Click the **Toggle auto connect of children** (  ) toolbar button.
- On the **Connection** menu, click **Auto Connect Matching Children**.

Notice that some of the input connectors on the target component have been highlighted by MapForce in orange, which indicates that these items are mandatory. To ensure the validity of the target XML file, provide values for the mandatory items as follows:


- Connect the `<category>` element in the source with the `<genre>` element in the target
- Connect the `<year>` element in the source with the `<publish_year>` element in the target

Finally, you need to supply a value to the `<last_updated>` element. If you move the mouse over its input connector, you can see that the element is of type `xs:dateTime`. Note that, for tips to be displayed, the **Show tips** (  ) toolbar button must be enabled.

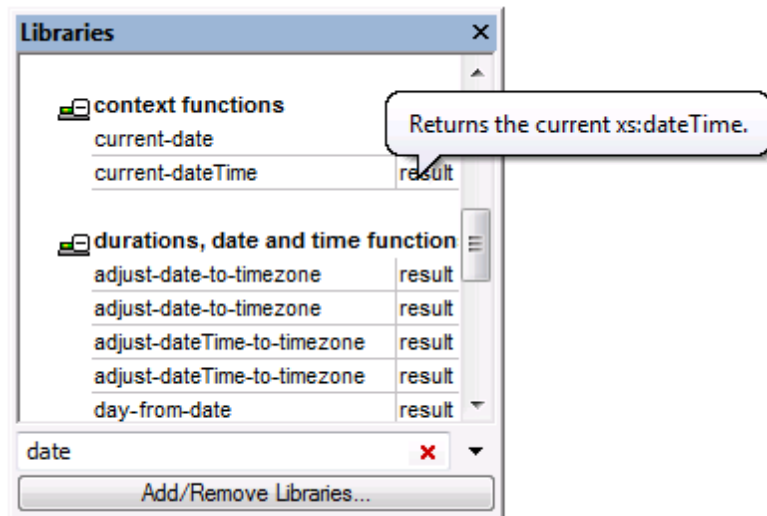



You can also make the data type of each item visible at all times, by clicking the **Show Data**



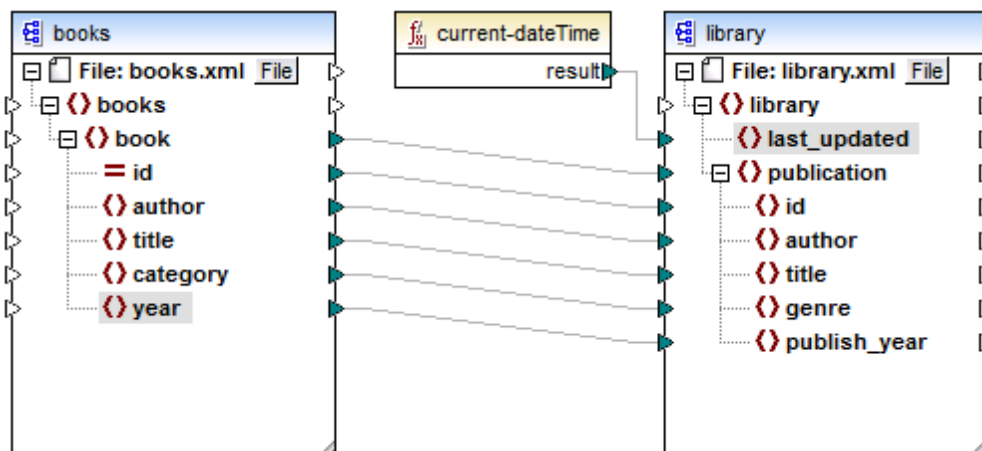
**Types** (  ) toolbar button.

You can get the current date and time (that is, the `xs:dateTime` value) by means of a date and time XSLT function. To find the XSLT function to the mapping, start typing "date" in the text box located in the lower part of the [Libraries window](#). Alternatively, double-click an empty area on the mapping and start typing "current-date".



As shown above, if you move the mouse over the "result" part of the function, you can see its description. For tips to be displayed, make sure that the **Show tips** (  ) toolbar button is enabled.

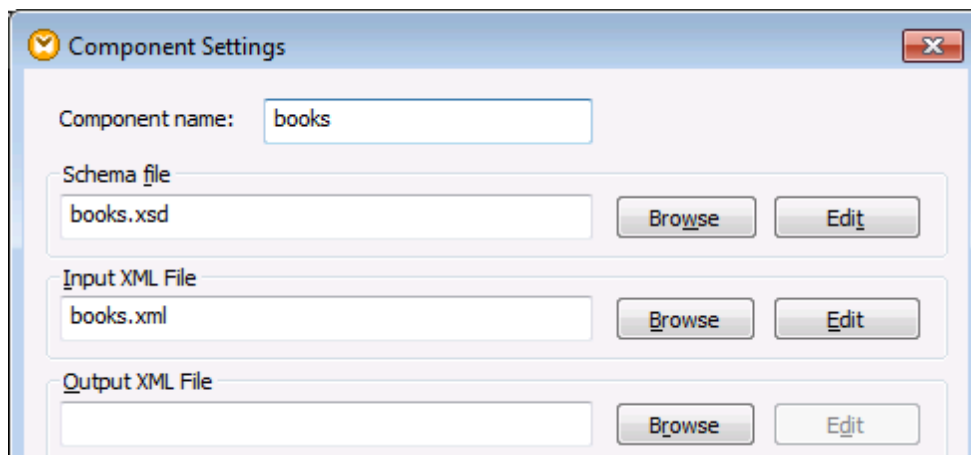
To add the function to the mapping, drag the function into the mapping pane, and connect its output to the input of the `<last_updated>` element.



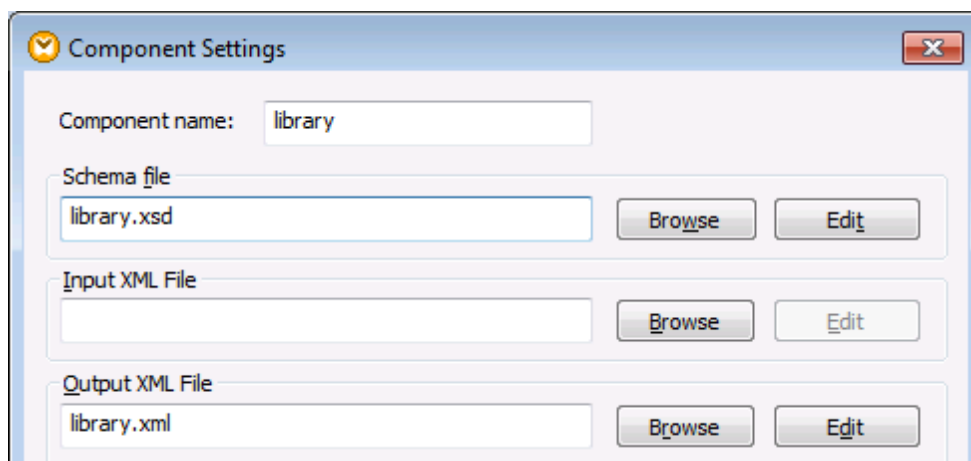
You have now created a MapForce mapping design (or simply a "mapping") which converts data from the **books.xml** instance file (having the **books.xsd** schema) to the new **library.xml** file (having the **library.xsd** schema). If you double-click the header of each component, you can view



these and other settings in the Component Settings dialog box, as shown below.




*Component settings for the source*



*Component settings for the target*

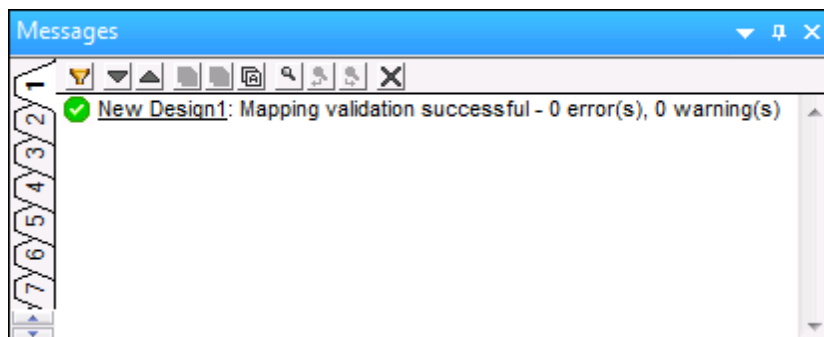
### Step 5: Validate and save the mapping

Validating a mapping is an optional step that enables you to see and correct potential mapping errors and warnings before you run the mapping. To check whether the mapping is valid, do one of the following:

- On the **File** menu, click **Validate Mapping**.
- Click the **Validate** (  ) toolbar button.


The Messages window displays the validation results:





*Messages window*

At this point, you might also want to save the mapping to a file. To save the mapping, do one of the following:

- On the **File** menu, click **Save**.
- Click the **Save** (  ) toolbar button.

For your convenience, the mapping created in this tutorial is available at the following path:

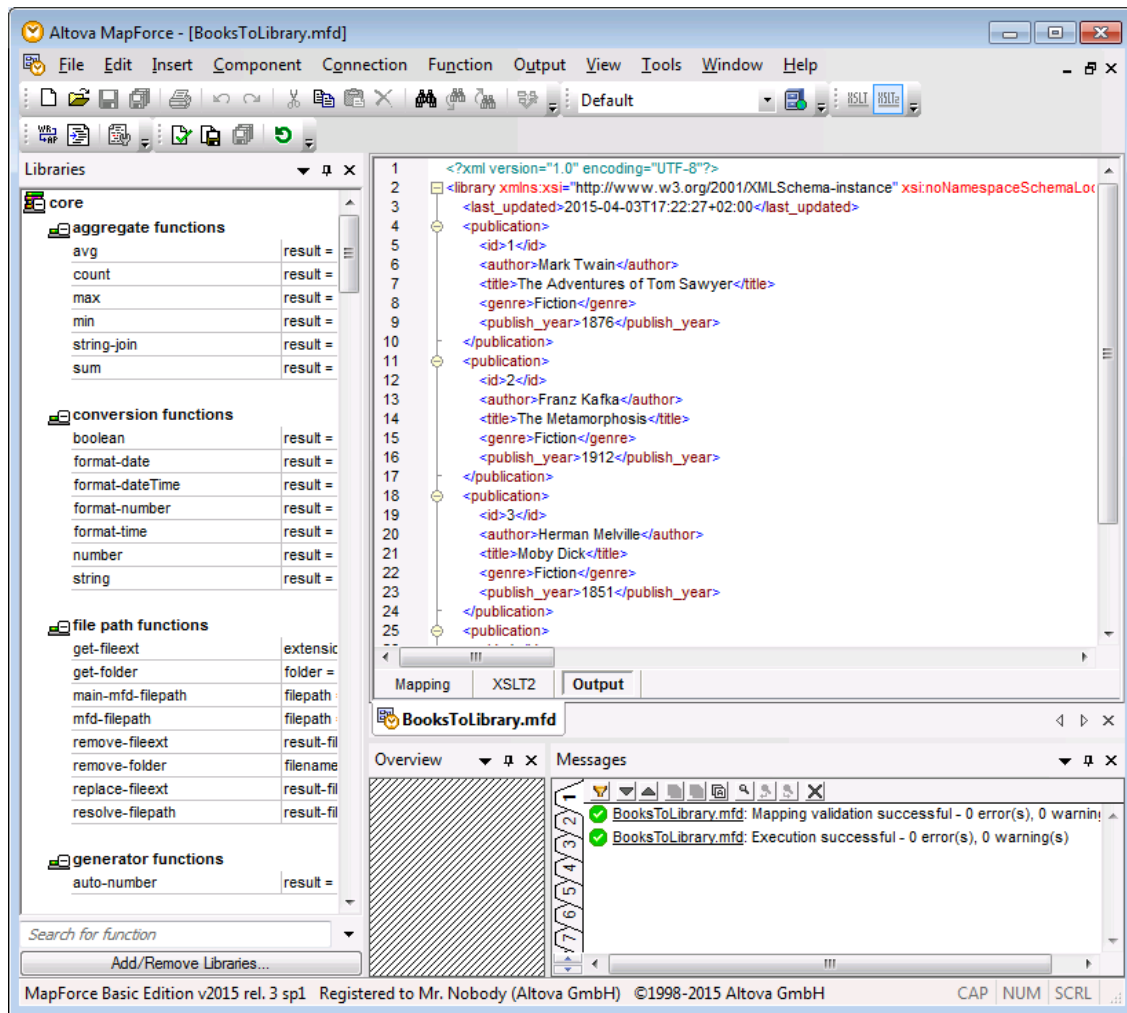
**<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\BooksToLibrary.mfd.**

Therefore, from this point onwards, you can either continue with the mapping file you created, or with the **BooksToLibrary.mfd** file.

### **Step 6: Preview the mapping result**


You can preview the result of the mapping directly in MapForce. To do this, click the **Output** button located in the lower part of the mapping pane. MapForce runs the transformation and displays the result of the mapping in the **Output** pane.





Output pane

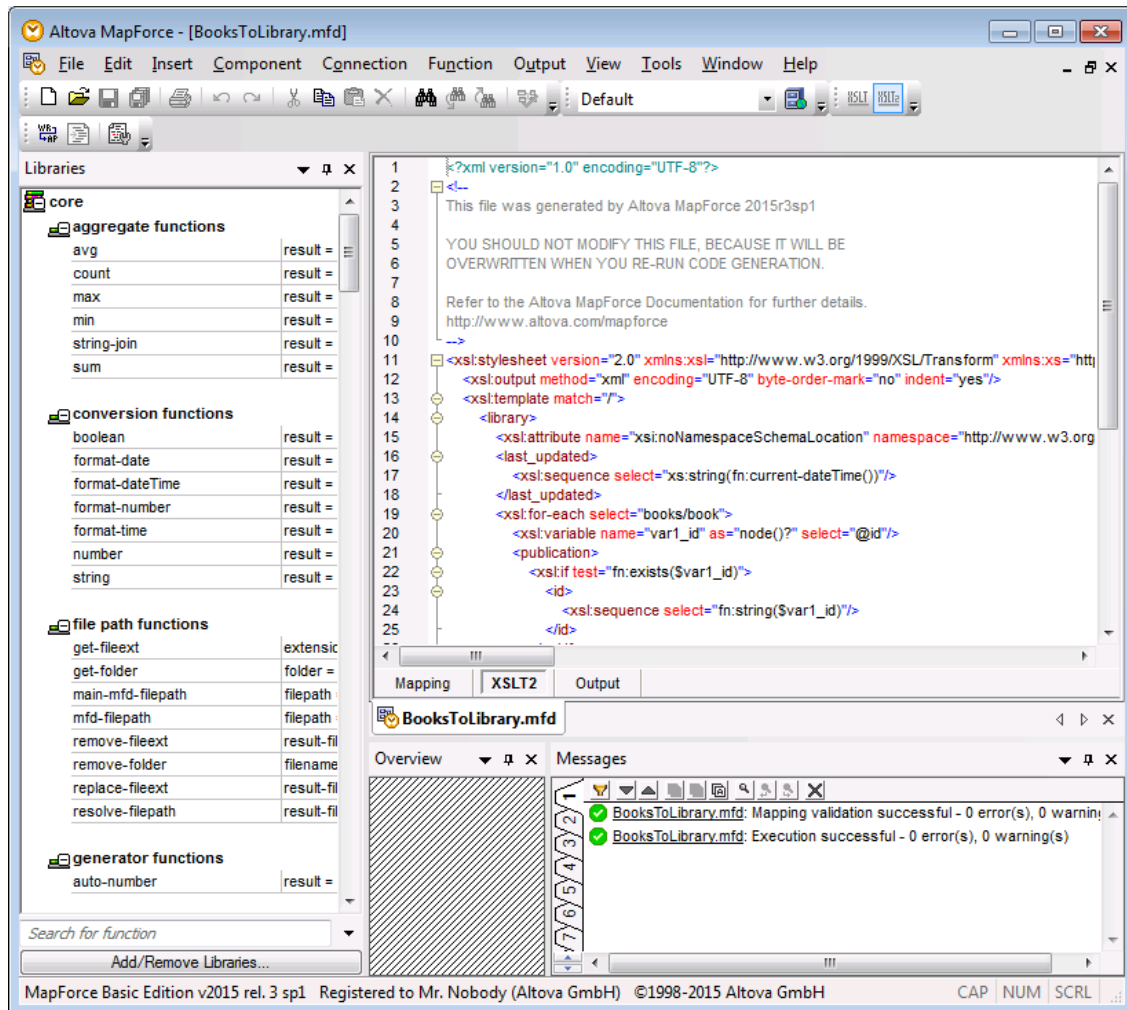
You can now see the result of the transformation in MapForce.

By default, the files displayed for preview in the **Output** pane are not written to the disk. Instead, MapForce creates temporary files. To save the file displayed in the **Output** pane to the disk, select the menu command **Output | Save Output File**, or click the **Save generated output** (  ) toolbar button.

To configure MapForce to write the output directly to final files instead of temporary, go to **Tools | Options | General**, and then select the **Write directly to final output files** check box. Note that enabling this option is not recommended while you follow this tutorial, because you may unintentionally overwrite the original tutorial files.

You can also preview the generated XSLT code that performs the transformation. To preview the code, click the **XSLT2** button located in the lower area of the mapping pane.





XSLT2 pane

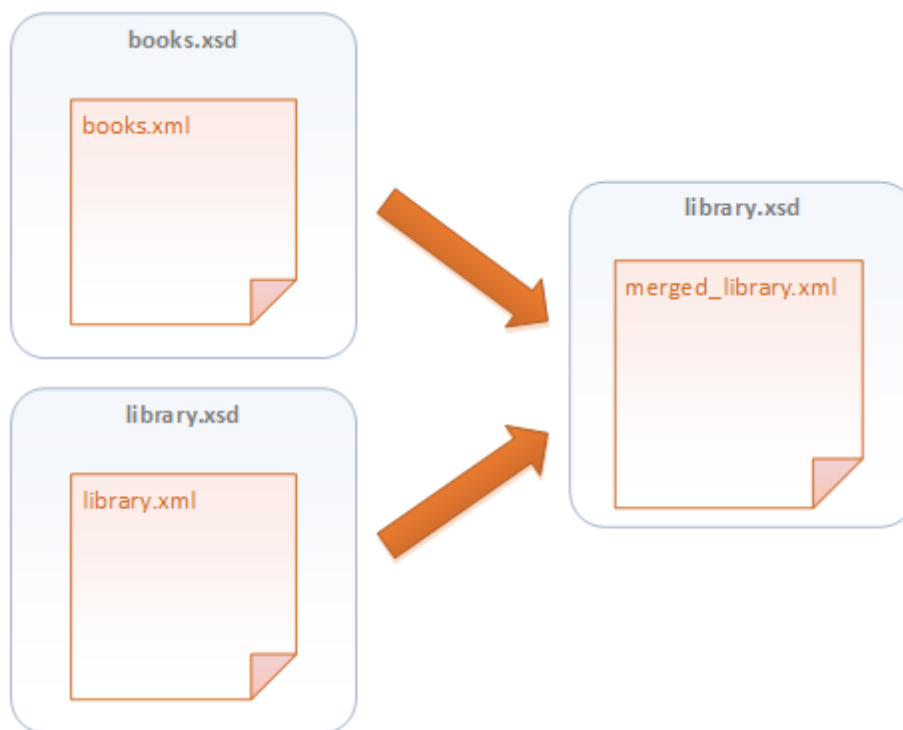
To generate and save the XSLT2 code to a file, select the menu item **File | Generate Code in | XSLT 2.0**. When prompted, select a folder where the generated code must be saved. After code generation completes, the destination folder includes the following two files:

1. An XSLT transformation file, named after the target schema (in this example, **MappingMaptolibrary.xslt**).
2. A **DoTransform.bat** file. The **DoTransform.bat** file enables you to run the XSLT transformation in RaptorXML Server (for more information, see <https://www.altova.com/raptorxml.html>).



## 3.2 Map Multiple Sources to One Target

In the previous tutorial, you have converted data from a source file (**books.xml**) to a target file (**library.xml**). The target file (**library.xml**) did not exist before running the mapping; it was generated by the mapping transformation. Let's now imagine a scenario where you already have some data in the **library.xml** file, and you want to merge this data with data converted from the **books.xml**. The goal in this tutorial is to design a mapping that generates a file called **merged\_library.xml**. The generated file will include data from two sources: the **books.xml** file and the **library.xml** file. Note that the files used as source (**books.xml** and **library.xml**) have different schemas. If the source files had the same schema, you could also merge their data using a different approach (see [Process and Generate Files Dynamically](#) ).



*Abstract model of the data transformation*

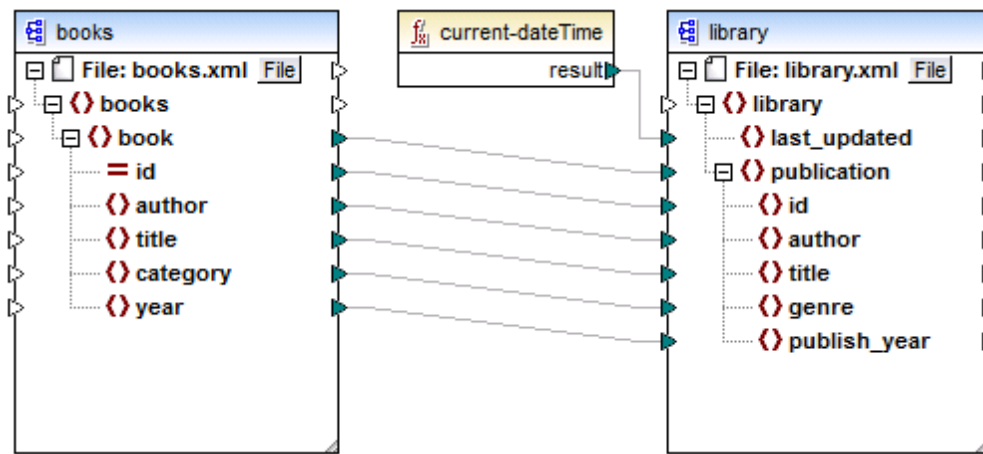
To achieve the required goal, let's take the following steps.

### Step 1: Prepare the mapping design file

This tutorial uses as starting point the **BooksToLibrary.mfd** mapping from the **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\** folder. You have already designed this mapping in the [Convert XML to New Schema](#) tutorial. To begin, open the **BooksToLibrary.mfd** file in MapForce, and save it with a new name.

Make sure to save the new mapping in the **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\** folder, because it references several files from it.



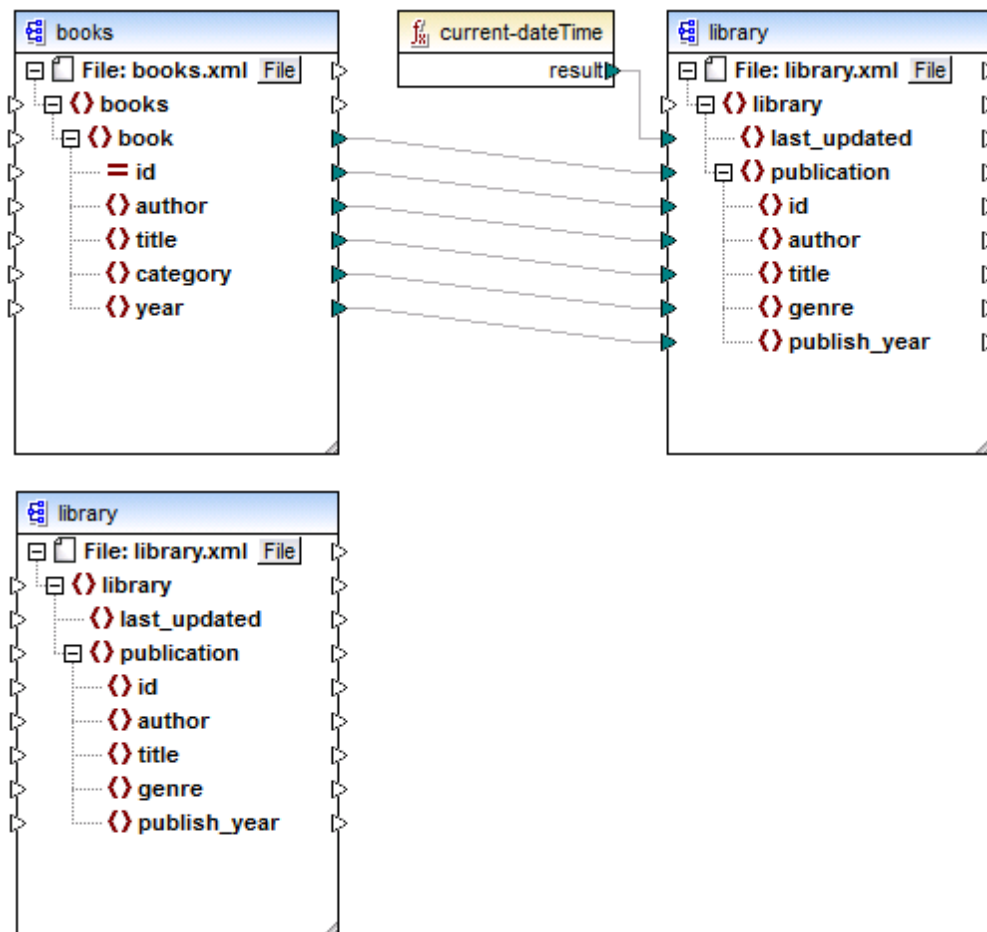


*BooksToLibrary.mfd (MapForce Basic Edition)*

## Step 2: Create a second source component

First, select the target component and copy it (press **Ctrl + C**), and then paste it (press **Ctrl + V**) into the same mapping. Click the header of the new component and drag it under the **books** component.





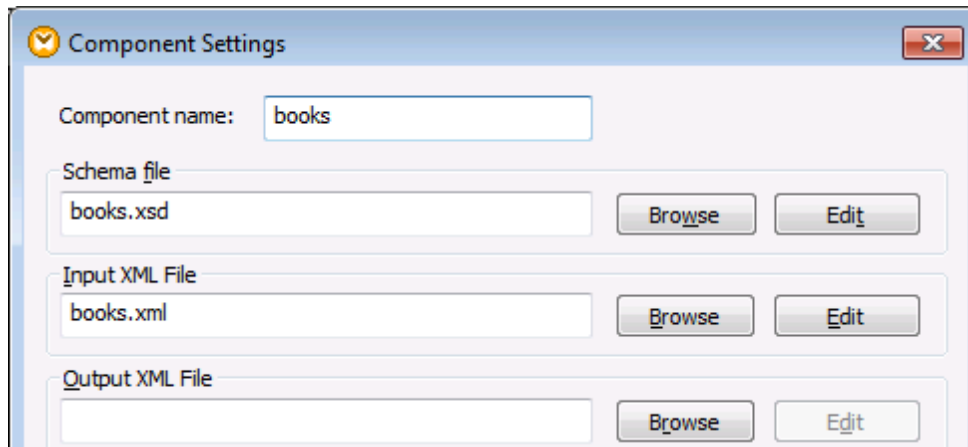
The mapping now has two source components: **books** and **library**, and one target component: **library**.

You can always move the mapping components in any direction (left, right, top, bottom). Nevertheless, placing a source component to the left of a target component will make your mapping easier to read and understand by others. This is also the convention for all mappings illustrated in this documentation, as well as in the sample mapping files accompanying your MapForce installation.

### Step 3: Verify and set the input/output files

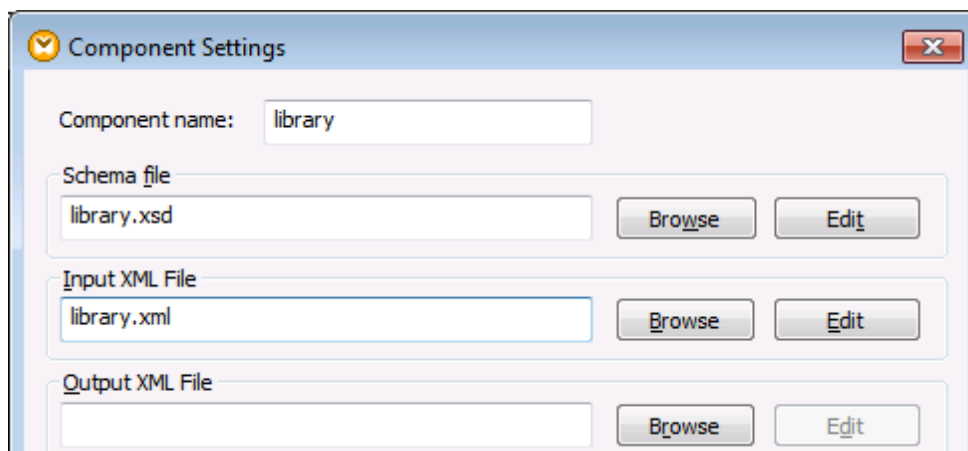
In the previous step, the new source component was copy-pasted from the target component, so it inherits the same settings. To ensure that the name input/output instance files are correctly set, double-click the header of each component, and, in the Component Settings dialog box, verify and change the name and the input/output files of each component as shown below.





The screenshot shows a 'Component Settings' dialog box with a yellow icon and a close button. It contains three sections: 'Component name' with the value 'books', 'Schema file' with 'books.xsd', and 'Input XML File' with 'books.xml'. Each section has a 'Browse' and an 'Edit' button. The 'Output XML File' section is empty and also has 'Browse' and 'Edit' buttons.

Components settings for the first source (**books**)



The screenshot shows a 'Component Settings' dialog box with a yellow icon and a close button. It contains three sections: 'Component name' with the value 'library', 'Schema file' with 'library.xsd', and 'Input XML File' with 'library.xml'. Each section has a 'Browse' and an 'Edit' button. The 'Output XML File' section is empty and also has 'Browse' and 'Edit' buttons.

Component settings for the second source (**library**)





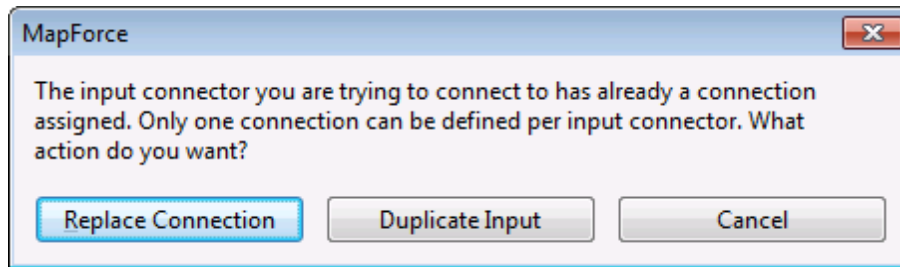
Component settings for the target (**merged\_library**)

As shown above, the first source component reads data from **books.xml**. The second source component reads data from **library.xml**. Finally, the target component outputs data to a file called **merged\_library.xml**.

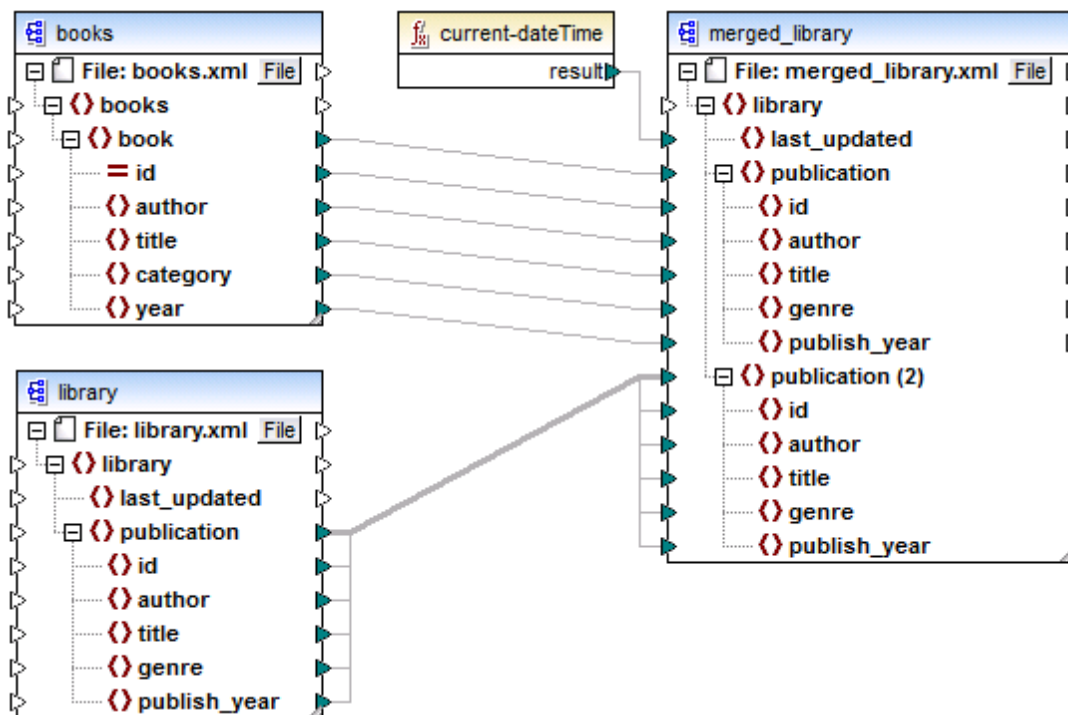
#### Step 4: Make the connections

To instruct MapForce to write data from the second source to the target, click the output connector (small triangle) of the `publications` item in the source **library** component and drag it to the input connector of the `publications` item in the target **library** component. Because the target input connector already has a connection to it, the following notification message appears.





In this particular tutorial, replacing the connection is not what we want to achieve; our goal is to map data from two sources. Therefore, click **Duplicate Input**. By doing so, you configure the target component to accept data from the new source as well. The mapping now looks as follows:



Notice that the `publication` item in the target component has now been duplicated. The new `publication(2)` node will accept data from the source **library** component. Importantly, even though the name of this node appears as `publication(2)` in the mapping, its name in the resulting XML file will be `publication`, which is the intended goal.

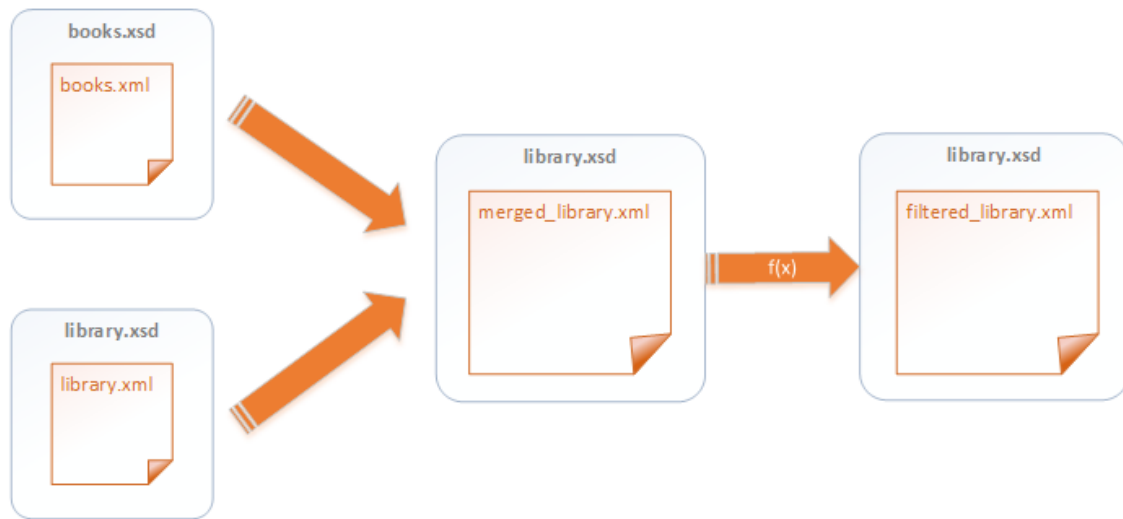
You can now click the **Output** button at the bottom of the mapping pane, and view the mapping result. You will notice that data from both **library.xml** and **books.xml** files has now been merged into the new **merged\_library.xml** file.



### 3.3 Work with Multiple Target Schemas

In the previous tutorial, [Map Multiple Sources to One Target](#), you have seen how to map data from multiple source schemas to a single target schema. You have also created a file called **merged\_library.xml**, which stores book records from two sources. Now let's assume that someone from another department has asked you to provide a subset of this XML file. Specifically, you must deliver an XML file that includes only the books published after 1900.

For convenience, you can modify the existing **MultipleSourcesToOneTarget.mfd** mapping so that, whenever required, you can generate both the complete XML library, and the filtered library.



*Abstract model of the data transformation*

In the diagram above, the data is first merged from two different schemas (**books.xsd** and **library.xsd**) into a single XML file called **merged\_library.xml**. Secondly, the data is transformed using a filtering function and passed further to the next component, which creates an XML file called **filtered\_library.xml**. The "intermediate" component acts both as data target and source. In MapForce, this technique is known as "chaining mappings", which is also the subject of this tutorial.

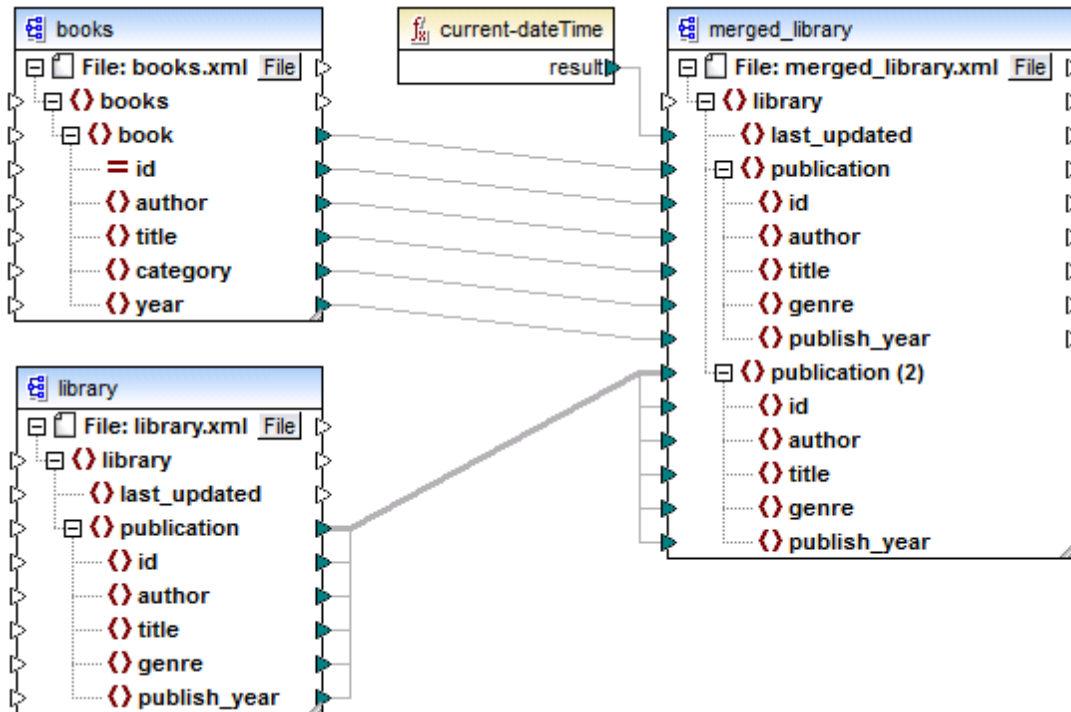
Our goal is to make it possible to generate at any time both the **merged\_library.xml** and the **filtered\_library.xml**. To achieve the goal, let's take the following steps.

#### Step 1: Prepare the mapping design file

This tutorial uses as starting point the **MultipleSourcesToOneTarget.mfd** mapping from the `<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\` folder. You have already designed this mapping in the [Map Multiple Sources to One Target](#) tutorial. To begin, open the **MultipleSourcesToOneTarget.mfd** file in MapForce, and save it with a new name.




Make sure to save the new mapping in the <Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\ folder, because it references several files from it.

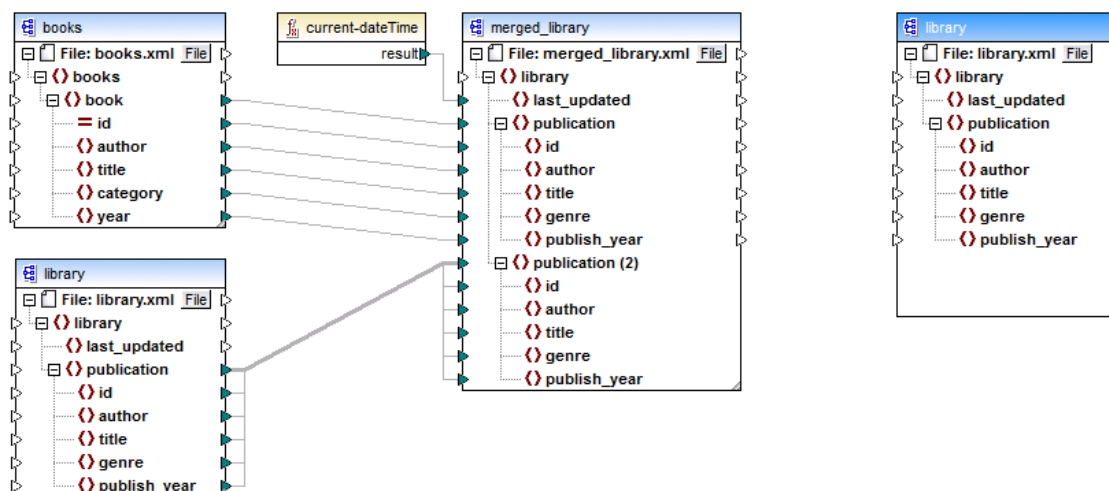


MultipleSourcesToOneTarget.mfd (MapForce Basic Edition)

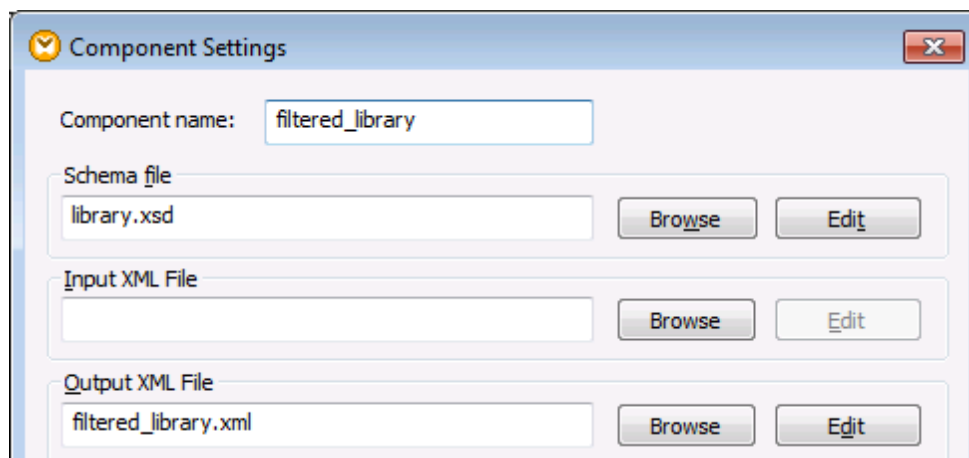
## Step 2: Add and configure the second target component

To add the second target component, click the **Insert XML Schema/File** () toolbar button, and open the **library.xsd** file located in the <Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\ folder. Click **Skip** when prompted to supply a sample instance file. The mapping now looks as follows:



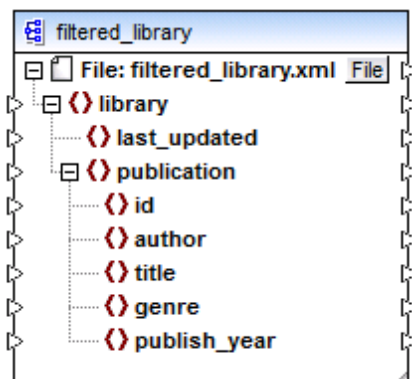


As shown above, the mapping now has two source components: **books** and **library**, and two target components. To distinguish between the target components, we will rename the second one to **filtered\_library**, and also set the name of the XML file that should be generated by it. To do this, double-click the header of the right-most component and edit the component settings as follows:



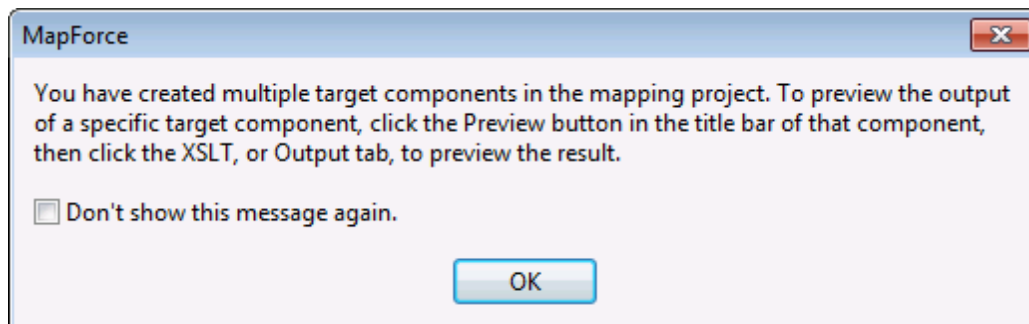
Notice that the new name of the component is **filtered\_library**, and the output XML file is named **filtered\_library.xml**.







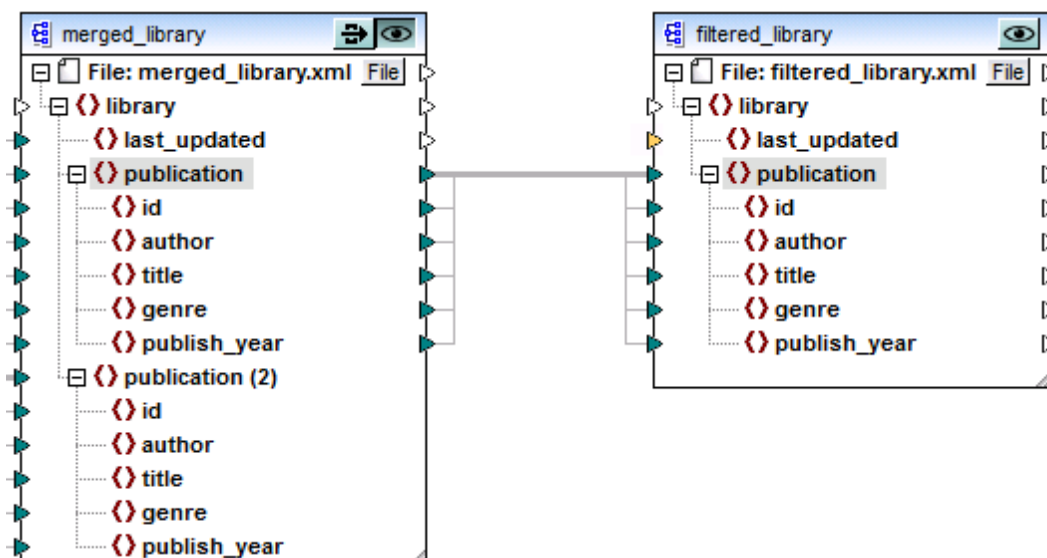
### Step 3: Make the connections

Create a connection from the item **publication** in the **merged\_library** to the item **publication** in the **filtered\_library**. When you do this, a notification message is displayed.



Click **OK**. Notice that new buttons are now available in the upper-right corner of both target components: **Preview** (  ) and **Pass-through** (  ). These buttons will be used and explained in the following steps.

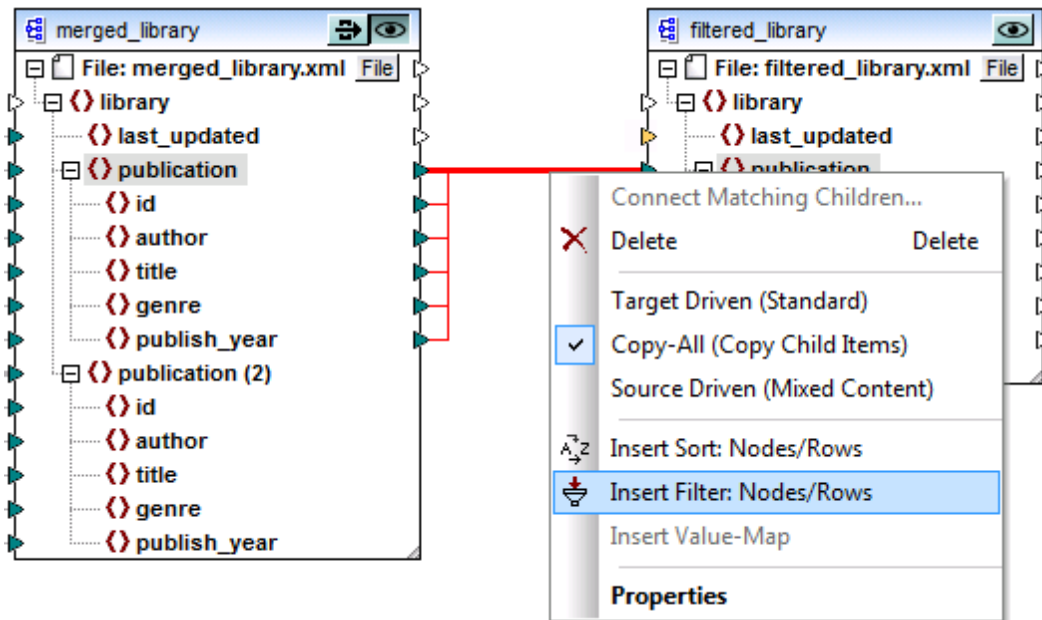




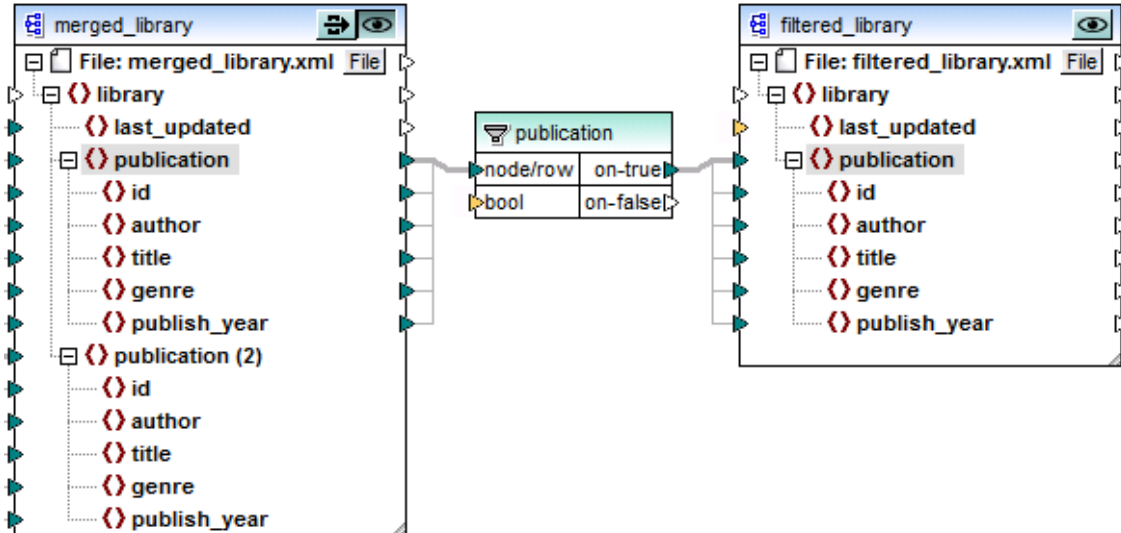
#### Step 4: Filter data


To filter data before supplying it to the **filtered\_library**, we will use a **Filter** component. To add a filter component, right-click the connection between **merged\_library** and **filtered\_library**, and select **Insert Filter: Nodes/Rows** from the context menu.



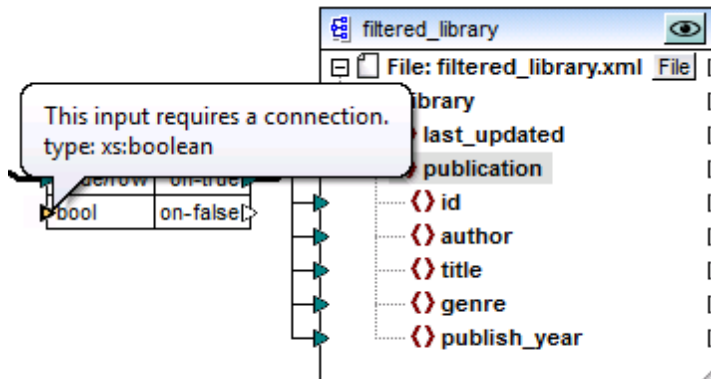


The filter component has now been added to the mapping.



As shown above, the bool input connector is highlighted in orange, which suggests that an input is required. If you move the mouse over the connector, you can see that an input of type `xs:boolean` is required. Note that, for tips to be displayed, the **Show tips** (  ) toolbar button must be enabled.

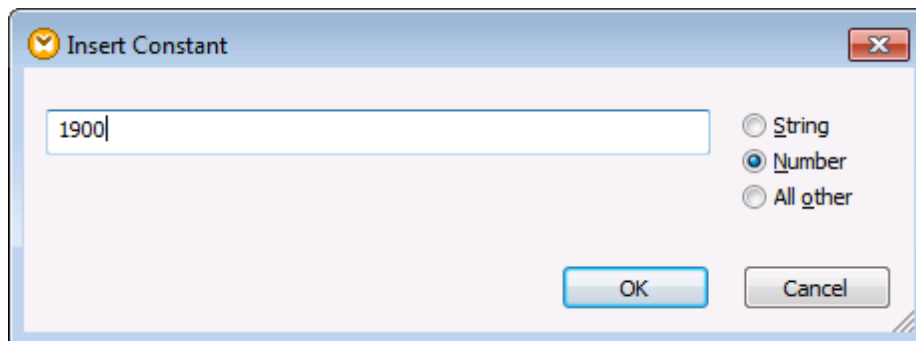




The filter component requires a condition that returns either `true` or `false`. When the Boolean condition returns `true`, data of the current **publication** sequence will be copied over to the target. When the condition returns `false`, data will not be copied.

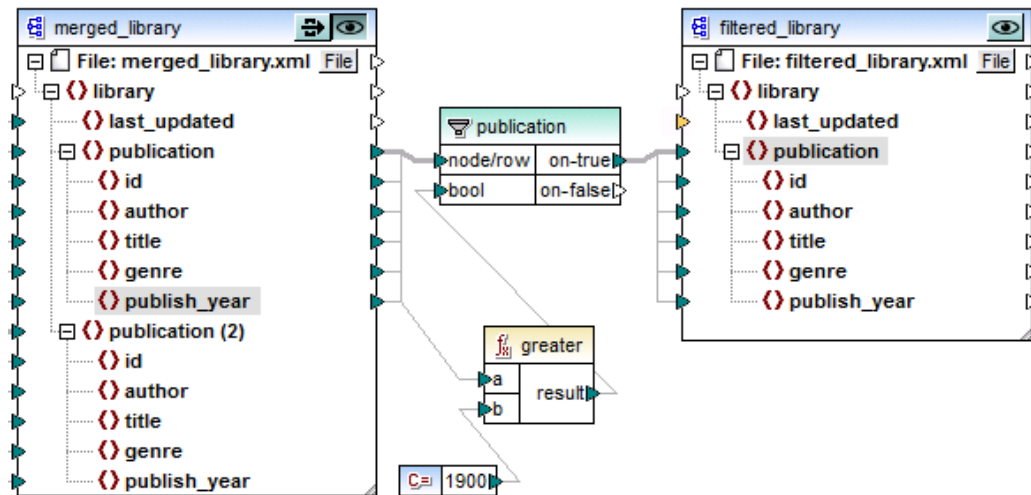
In this tutorial, the required condition is to filter all books which were published after 1900. To create the condition, do the following:

1. Add a constant of numeric type having the value "1900" (On the **Insert** menu, click **Constant**). Choose **Number** as type.





2. In the Libraries window, locate the function **greater** and drag it to the mapping pane.
3. Make the mapping connections to and from the function **greater** as shown below. By doing this, you are instructing MapForce: "When `publish_year` is greater than 1900, copy the current `publication` source item to the `publication` target item".






### Step 5: Preview and save the output of each target component



You are now ready to preview and save the output of both target components. When multiple target components exist in the same mapping, you can choose which one to preview by clicking the **Preview** (  ) button. When the **Preview** button is in a pressed state (  ), it indicates that that specific component is currently enabled for preview (and this particular component will generate the output in the Preview pane). Only one component at a time can have the preview enabled.


Therefore, when you want to view and save the output of the **merged\_library** (that is, the "intermediate") component, do the following:

1. Click the **Preview** button (  ) on the **merged\_library** component.
2. Click the **Output** button at the bottom of the mapping pane.
3. On the **Output** menu, click **Save Output File** if you want to save the output to a file.

When you want to view and save the output of the **filtered\_library** component :

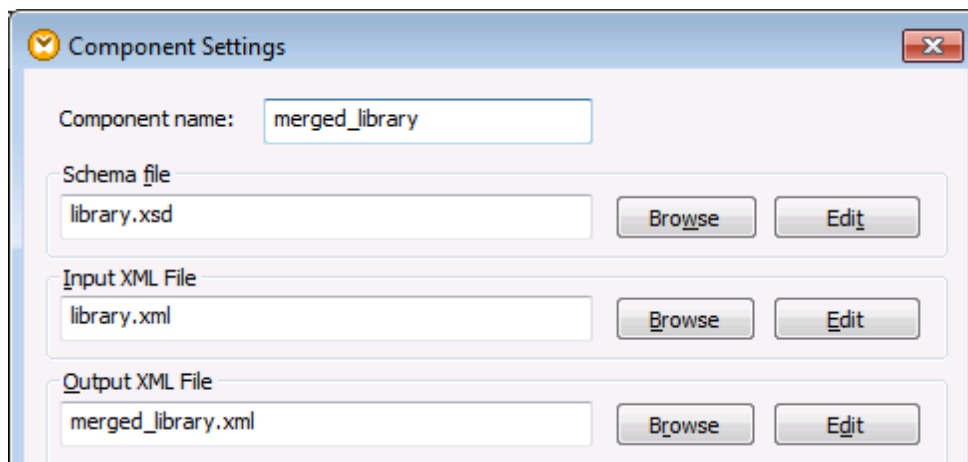
1. Click the **Pass-through** button (  ) on the **merged\_library** component.
2. Click the **Preview** button (  ) on the **filtered\_library** component.
3. Click the **Output** button at the bottom of the mapping pane.
4. On the **Output** menu, click **Save Output File** if you want to save the output to a file.

Notice the **Pass-through** (  ) button—clicking or not clicking it makes a big difference in any mapping which has multiple target components, including this one. When this button is in a pressed state (  ), MapForce lets data pass through the intermediate component, so that you can preview the result of the entire mapping.

Release the button (  ) if you want to preview only the portion of the mapping between the **merged\_library** and the **filtered\_library**. In the latter case, an error will be generated. This behavior is expected, because the intermediate component does not have a valid input XML file from which it should read data. To solve the problem, double-click the header of the component



and edit so as to supply a valid input XML file, as shown below:



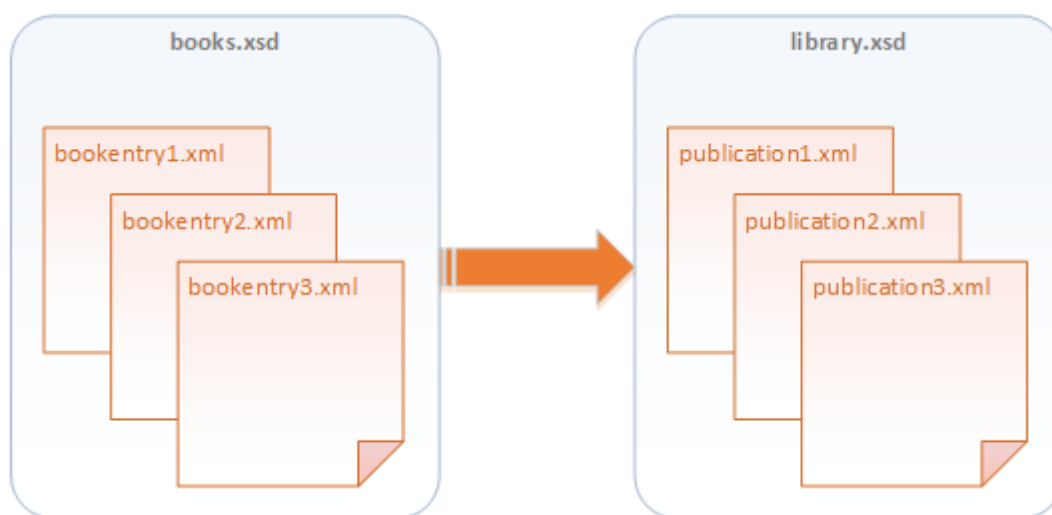
You have now finished designing a mapping which has multiple target components, and you can view and save the output of each target, which was the intended goal of this tutorial. For further information about working with pass-through components, see [Chained mappings / pass-through components](#).



## 3.4 Process and Generate Files Dynamically

This tutorial shows you how to read data from multiple source XML files and write it to multiple target files in the same transformation. To illustrate this technique, we will now create a mapping with the following goals:

1. Read data from multiple XML files in the same directory.
2. Convert each file to a new XML schema.
3. For each source XML file, generate a new XML target file under the new schema.
4. Strip the XML and namespace declaration from the generated files.



*Abstract model of the data transformation*

We will use three source XML files as example. The files are located in the **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\** folder, and they are named **bookentry1.xml**, **bookentry2.xml**, and **bookentry3.xml**. Each of the three files stores a single book.

```
<?xml version="1.0" encoding="UTF-8"?>
<books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="books.xsd">
  <book id="1">
    <author>Mark Twain</author>
    <title>The Adventures of Tom Sawyer</title>
    <category>Fiction</category>
    <year>1876</year>
  </book>
</books>
```

*bookentry1.xml*



```
<?xml version="1.0" encoding="UTF-8"?>
<books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="books.xsd">
  <book id="2">
    <author>Franz Kafka</author>
    <title>The Metamorphosis</title>
    <category>Fiction</category>
    <year>1912</year>
  </book>
</books>
```

*bookentry2.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="books.xsd">
  <book id="3">
    <author>Herman Melville</author>
    <title>Moby Dick</title>
    <category>Fiction</category>
    <year>1851</year>
  </book>
</books>
```

*bookentry3.xml*

The source XML files use the **books.xsd** schema available in the following folder: **<Documents> \Altova\MapForce2018\MapForceExamples\Tutorial\**. To convert the source files to a new XML schema, we will use the **library.xsd** schema (available in the same folder). After the transformation, the mapping will generate three files according to this new schema (see the code listings below). We will also configure the mapping so that the name of the generated files will be: **publication1.xml**, **publication2.xml**, and **publication3.xml**. Notice that the XML declaration and the namespace declaration must be stripped.

```
<library>
  <publication>
    <id>1</id>
    <author>Mark Twain</author>
    <title>The Adventures of Tom Sawyer</title>
    <genre>Fiction</genre>
    <publish_year>1876</publish_year>
  </publication>
</library>
```

*publication1.xml*



```
<library>
  <publication>
    <id>2</id>
    <author>Franz Kafka</author>
    <title>The Metamorphosis</title>
    <genre>Fiction</genre>
    <publish_year>1912</publish_year>
  </publication>
</library>
```

*publication2.xml*

```
<library>
  <publication>
    <id>3</id>
    <author>Herman Melville</author>
    <title>Moby Dick</title>
    <genre>Fiction</genre>
    <publish_year>1851</publish_year>
  </publication>
</library>
```

*publication3.xml*

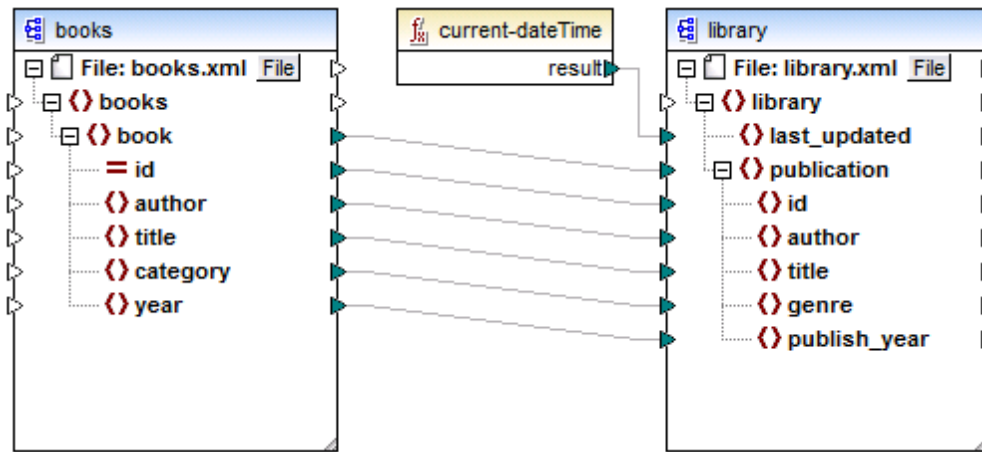
To achieve the goals, let's take the following steps.

### Step 1: Prepare the mapping design file

This tutorial uses as starting point the **BooksToLibrary.mfd** mapping from the **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\** folder. You have already designed this mapping in the [Convert XML to New Schema](#) tutorial. To begin, open the **BooksToLibrary.mfd** file in MapForce, and save it with a new name, in the same folder.

Make sure to save the new mapping in the **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\** folder, because it references several files from it.

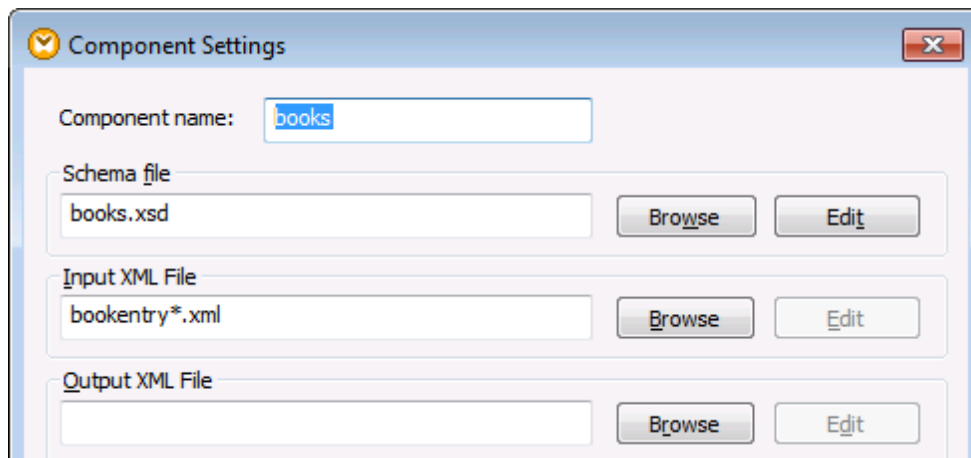




BooksToLibrary.mfd (MapForce Basic Edition)

## Step 2: Configure the input

To instruct MapForce to process multiple XML instance files, double-click the header of the source component. In the Component Settings dialog box, enter **bookentry\*.xml** as input file.



Component Settings dialog box



The asterisk ( \* ) wildcard character in the file name instructs MapForce to use as mapping input all the files that have the **bookentry-** prefix. Because the path is a relative one, MapForce will look for all **bookentry-** files in the same directory as the mapping file. Note that you could also enter an absolute path if necessary, while still using the \* wildcard character.

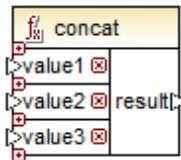
## Step 3: Configure the output

To create the file name of each output file, we will use the **concat** function. This function concatenates (joins) all the values supplied to it as argument.

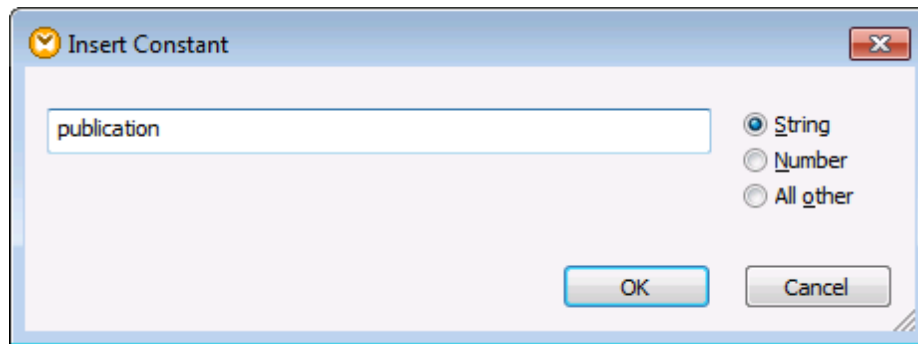


To build the file name using the **concat** function:

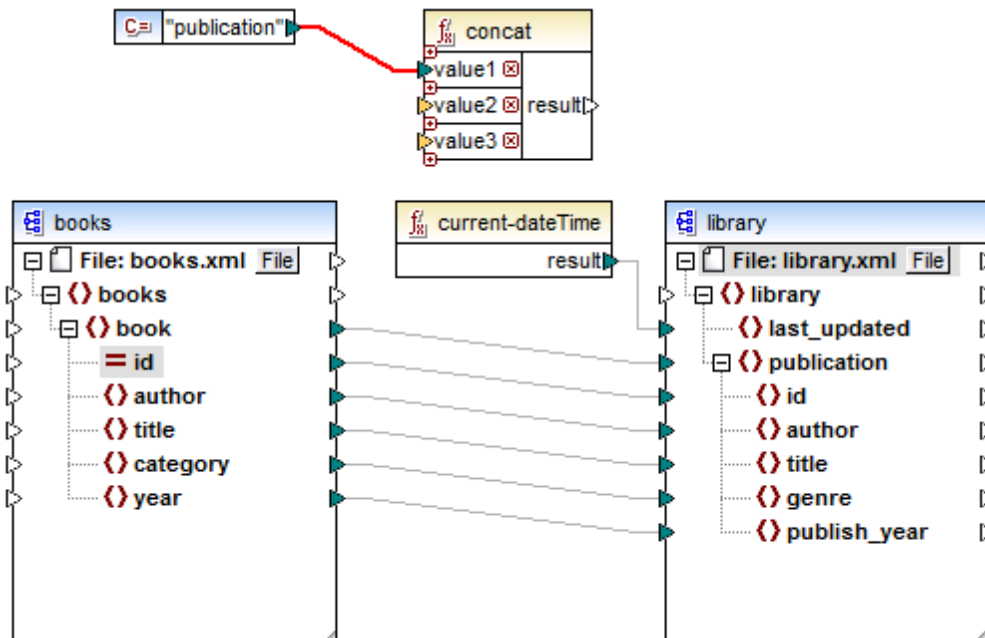
1. Search for the **concat** function in the Libraries window and drag it to the mapping area. By default, this function is added to the mapping with two parameters; however, you can add new parameters if necessary. Click the **Add parameter** (  ) symbol inside the function component and add a third parameter to it. Note that clicking the **Delete parameter** (  ) symbol deletes a parameter.



2. Insert a constant (on the **Insert** menu, click **Constant**). When prompted to supply a value, enter "publication" and leave the **String** option unchanged.

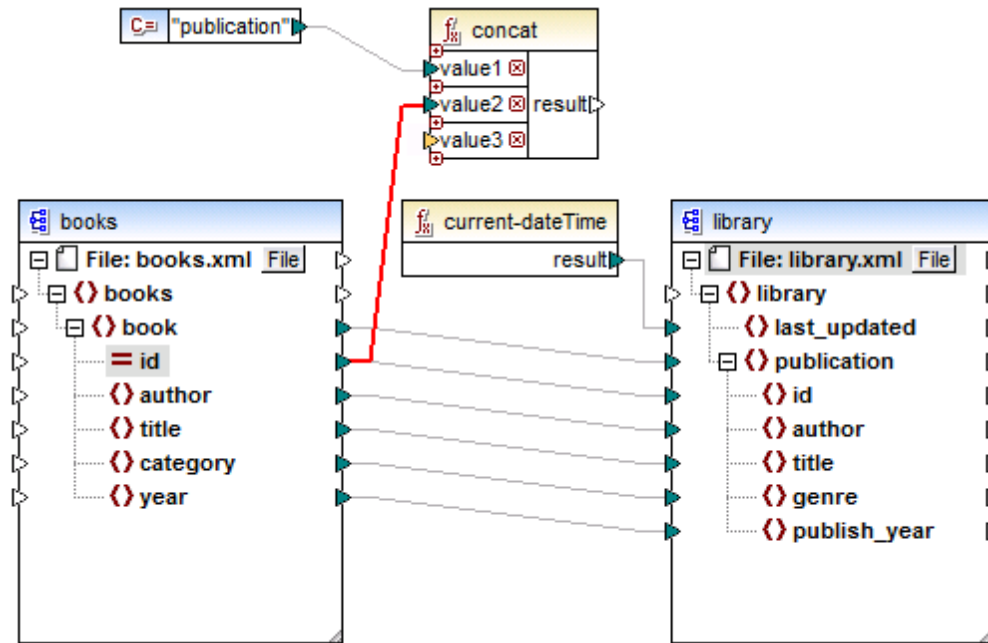


3. Connect the constant with **value1** of the **concat** function.

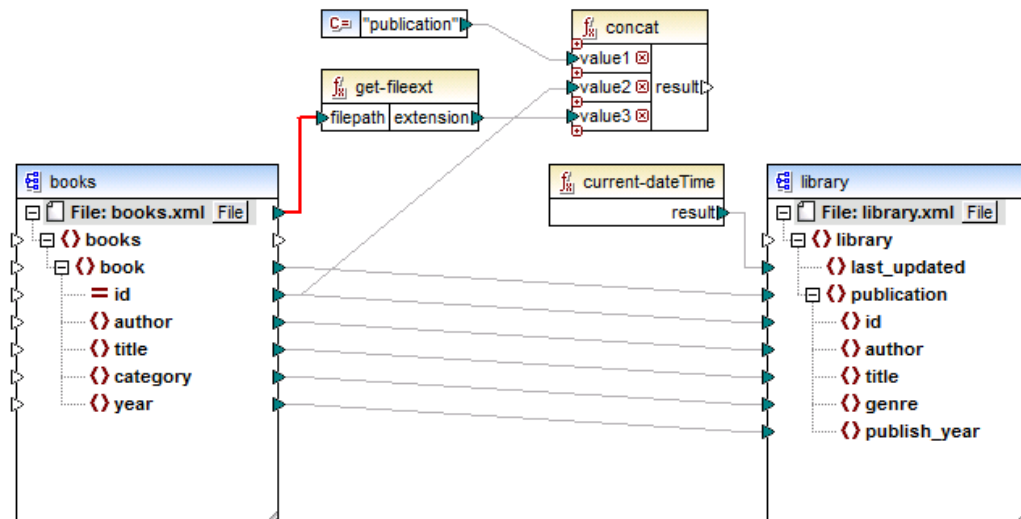


4. Connect the **id** attribute of the source component with **value2** of the **concat** function.





5. Search for the `get-fileext` function in the Libraries window and drag it to the mapping area. Create a connection from the top node of the source component (**File: books.xml**) to the **filepath** parameter of this function. Then create a connection from the result of the `get-fileext` function to **value3** of the `concat` function. By doing this, you are extracting only the extension part (in this case, .xml) from the source file name.

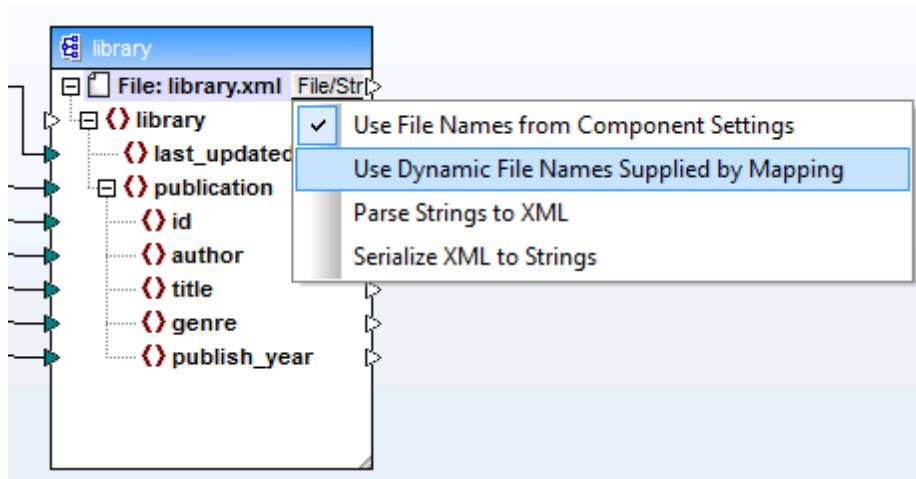


So far, you have provided as parameters to the `concat` function the three values which, when joined together, will create the generated file name (for example, **publication1.xml**):



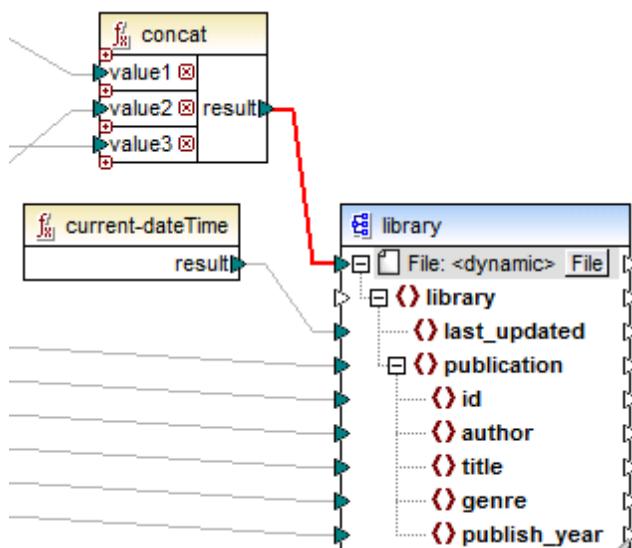
Part	Example
The constant "publication" supplies the constant string value "publication".	publication
The attribute <code>id</code> of the source XML file supplies a unique identifier value for each file. This is to prevent all files from being generated with the same name.	1
The <code>get-fileext</code> function returns the extension of the file name to be generated.	.xml

You can now instruct MapForce to actually build the file name when the mapping runs. To do this, click the **File** ( `File` ) or **File/String** ( `File/String` ) button of the target component and select **Use Dynamic File Names Supplied by Mapping**.

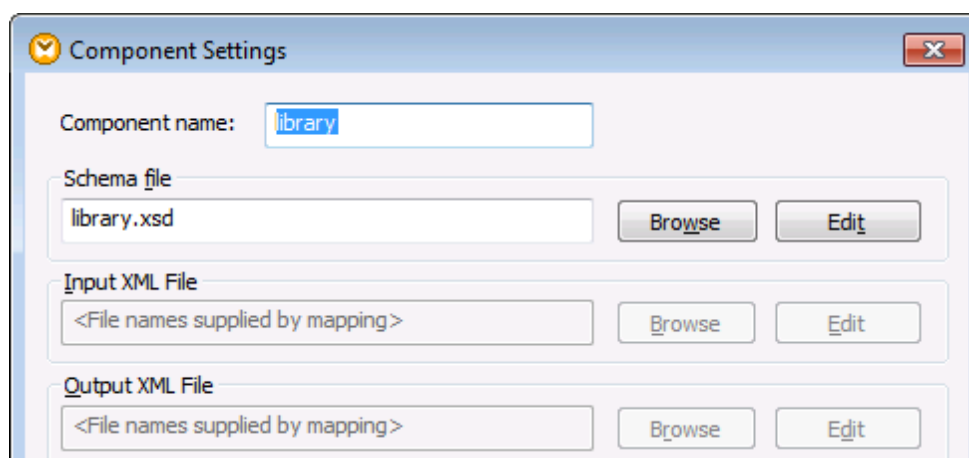


You have now instructed MapForce to generate the instance files dynamically, with whatever name will be provided by the mapping. In this particular example, the name is created by the `concat` function; therefore, we will connect the result of the `concat` function with the **File:** `<dynamic>` node of the target component.





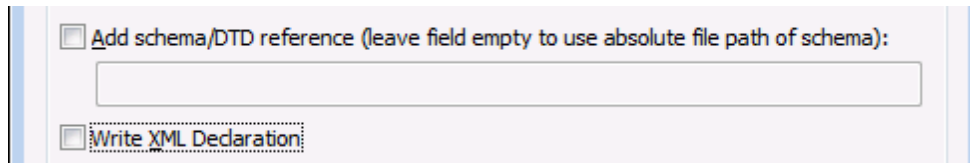
If you double-click the target component header at this time, you will notice that the **Input XML File** and **Output XML File** text boxes are disabled, and their value shows **<File names supplied by the mapping>**.



This serves as an indication that you have supplied the instance file names dynamically from a mapping, so it is no longer relevant to define them in the component settings.

Finally, you need to strip the XML namespace and schema declaration from the target. To achieve this, clear the selection from the **Add schema/DTD reference...** and **Write XML Declaration** check boxes on the Component Settings dialog box.

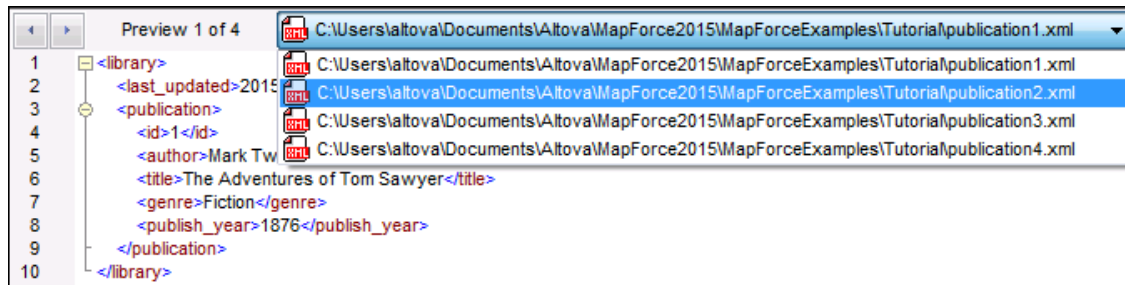




☐ Add schema/DTD reference (leave field empty to use absolute file path of schema):

☐ Write XML Declaration

You can now run the mapping and see the result, as well as the name of generated files. This mapping generates multiple output files. You can navigate through the output files using the left and right buttons in the upper left corner of the output pane, or by picking a file from the adjacent drop-down list.



Preview 1 of 4

1 <library>

2 <last\_updated>2015

3 <publication>

4 <id>1</id>

5 <author>Mark Tw</author>

6 <title>The Adventures of Tom Sawyer</title>

7 <genre>Fiction</genre>

8 <publish\_year>1876</publish\_year>

9 </publication>

10 </library>

File list:

- C:\Users\altova\Documents\Altova\MapForce2015\MapForceExamples\Tutorial\publication1.xml
- C:\Users\altova\Documents\Altova\MapForce2015\MapForceExamples\Tutorial\publication2.xml
- C:\Users\altova\Documents\Altova\MapForce2015\MapForceExamples\Tutorial\publication3.xml
- C:\Users\altova\Documents\Altova\MapForce2015\MapForceExamples\Tutorial\publication4.xml



# Chapter 4

---

## Common Tasks



## 4 Common Tasks


This section describes common MapForce tasks and concepts, such as working with mappings, components, and connections.



## 4.1 Working with Mappings

A MapForce mapping design (or simply "mapping") is the visual representation of how data is to be transformed from one format to another. A mapping consists of [components](#) that you add to the MapForce mapping area in order to create your data transformations (for example, convert XML documents from one schema to another). A valid mapping consists of one or several [source components](#) connected to one or several [target components](#). You can run a mapping and preview its result directly in MapForce. You can generate code and execute it externally. You can also compile a mapping to a MapForce execution file and automate mapping execution using MapForce Server or FlowForce Server. MapForce saves mappings as files with .mfd extension.

### To create a new mapping:

1. Do one of the following:
  - On the **File** menu, click **New**.
  - Click the **New** (  ) toolbar button.

Your mapping is now created; however, it does not yet do anything because it is empty. A mapping requires at least two connected [components](#) to become valid, so the next step is to add components to the mapping (see [Adding Components to the Mapping](#) ) and draw connections between components (see [Working with Connections](#) ).

### 4.1.1 Adding Components to the Mapping

In MapForce, the term "component" is what represents visually the structure (schema) of your data, or how data is to be transformed (functions). Components are the central building pieces of any [mapping](#). On the mapping area, components appear as rectangles. The following are examples of MapForce components:

- Constants
- Filters
- Conditions
- Function components
- EDI documents (UN/EDIFACT, ANSI X12, HL7)
- Excel 2007+ files
- Simple [input components](#)
- Simple [output components](#)
- XML Schemas and DTDs

### To add a component to the mapping, do one of the following:

- On the **Insert** menu, click the option relevant for the component type you wish to add (for example, **XML Schema/File**).
- Drag a file from Windows File Explorer onto the mapping area. Note that this operation is possible only for compatible file-based components.
- Click the relevant button on the Insert Component toolbar.





*Insert Component toolbar (MapForce Enterprise Edition)*

Each component type has specific purpose and behavior. For component types where that is necessary, MapForce walks you through the process by displaying contextual wizard steps or dialog boxes. For example, if you are adding an XML schema, a notification dialog box prompts you to optionally select an instance file as well.

For an introduction to components, see [Working with Components](#). For specific information about each technology supported as mapping source or target, see [Data Sources and Targets](#). For information about MapForce built-in components used to store data temporarily or transform it (such as filtering or sorting), see [Designing Mappings](#).

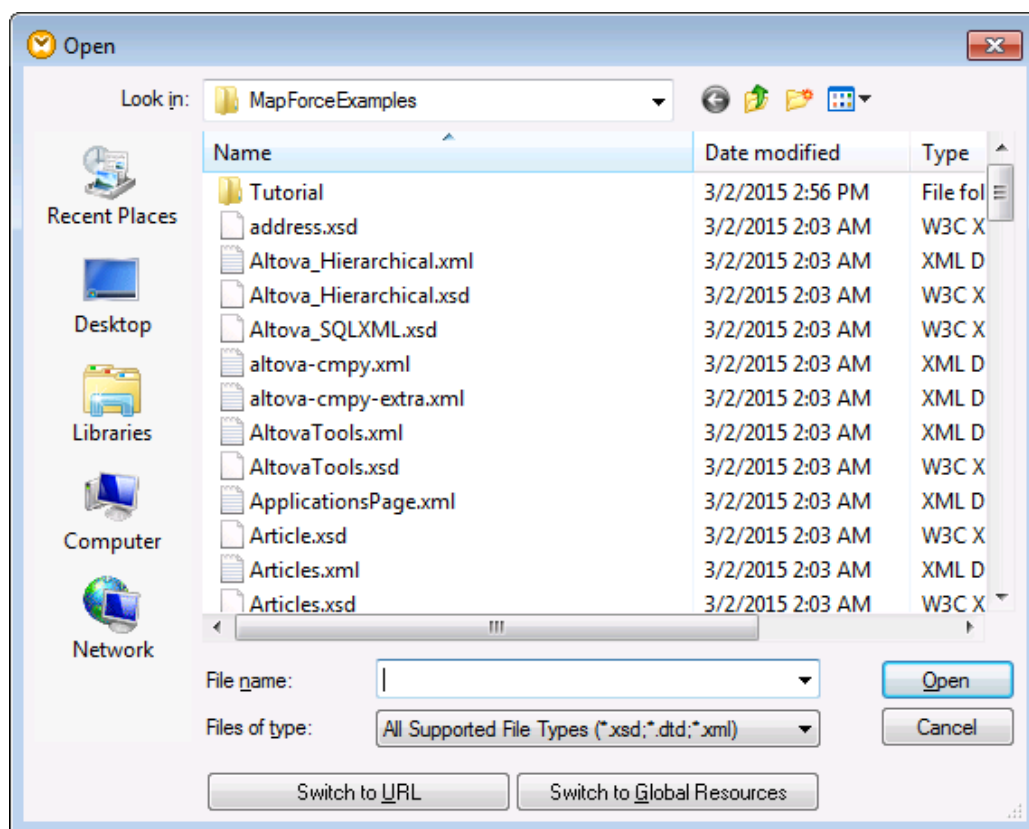
### 4.1.2 Adding Components from a URL

In addition to adding local files as mapping components, you can also add files from a URL. Note that this operation is supported when you add a component as source component (that is, your mapping reads data from the remote file). The supported protocols are HTTP, HTTPS, and FTP.

#### To add a component from a URL:

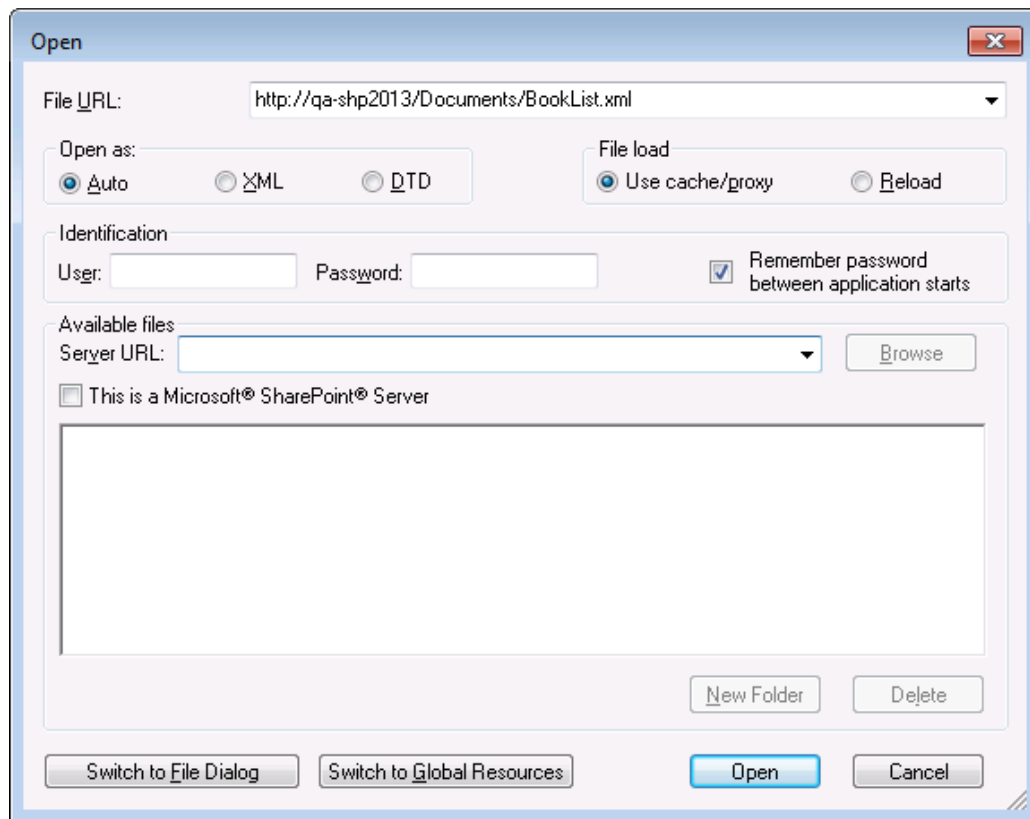
1. On the **Insert** menu, select the type of the component type you wish to add (for example, **XML Schema/File**).
2. On the **Open** dialog box, click **Switch to URL**.





3. Enter the URL of the file in the **File URL** text box, and click **Open**.





Make sure that the file type in the **File URL** text box is the same as the file type you specified in step 1.

If the server requires password authentication, you will be prompted to enter the user name and password. If you want the user name and password to be remembered next time you start MapForce, enter them in the Open dialog box and select the **Remember password between application starts** check box.

The **Open As** setting defines the grammar for the parser when opening the file. The default and recommended option is **Auto**.

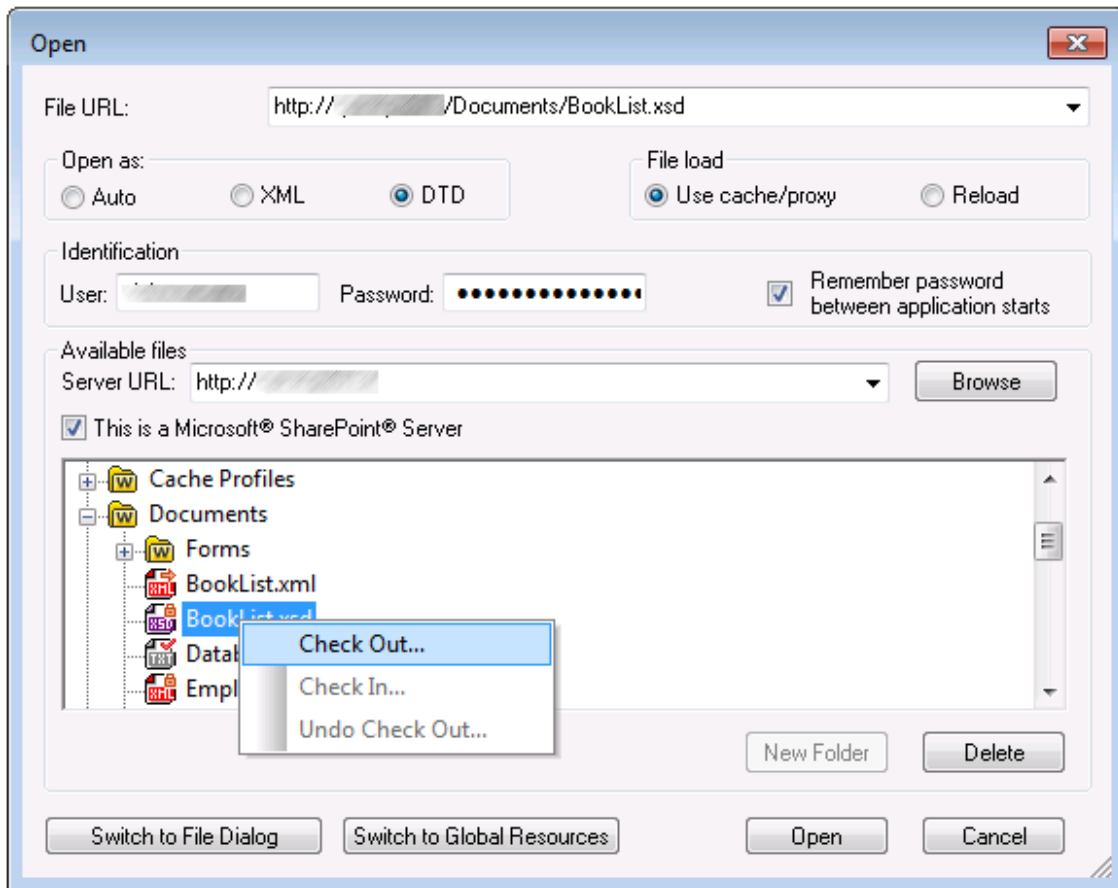
If the file you are loading is not likely to change, select the **Use cache/proxy** option to cache data and speed up loading the file. Otherwise, if you want the file to be reloaded each time when you open the mapping, select **Reload**.

For servers with Web Distributed Authoring and Versioning (WebDAV) support, you can browse files after entering the server URL in the **Server URL** text box and clicking **Browse**. Although the preview shows all file types, make sure that you choose to open the same file type as specified in step 1 above; otherwise, errors will occur.

If the server is a Microsoft SharePoint Server, select the **This is a Microsoft SharePoint Server** check box. Doing so displays the check-in or check-out state of the file in the preview area. If you want to make sure that no one else can edit the file on the server while you are using it in



MapForce to read data from it, right-click the file and select **Check Out**. To check in any file that was previously checked out by you, right-click the file and select **Check In**.



Open dialog box (in Switch to URL mode)

### 4.1.3 Selecting a Transformation Language

You can choose one of the following as data transformation language:

- XSLT 1.0
- XSLT 2.0



To select a transformation language, do one of the following:


- On the **Output** menu, click the name of the language you wish to use for transformation.
- Click the name of the language in the Language Selection toolbar.



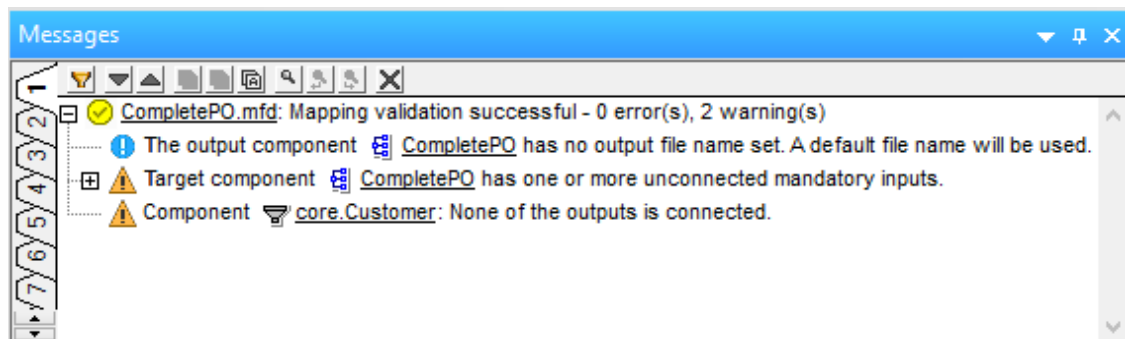
### 4.1.4 Validating Mappings

MapForce validates mappings automatically, when you click the **Output** tab to preview the transformation result. You can also validate a mapping explicitly, before attempting to preview its result. This helps you identify and correct potential mapping errors and warnings before the mapping is run. Note that running a mapping may generate additional runtime errors or warnings depending on the processed data, for example, when values mapped to attributes are overwritten.

To validate a mapping explicitly, do one of the following:




- On the **File** menu, click **Validate Mapping**.
- Click the **Validate** (  ) toolbar button.

The Messages window displays the validation results, for example:





*Messages window*


When you validate a mapping, MapForce checks for the validity of the mapping (such as incorrect or missing connections, unsupported component kinds), and the validation result is then displayed in the Messages window with one of the following status icons:

Icon	Meaning
	Validation has completed successfully.
	Validation has completed with warnings.
	Validation has failed.

The Message window may additionally display any of the following message types: information messages, warnings, and errors.

Icon	Meaning
	Denotes an information message. Information messages do not stop the mapping execution.
	Denotes a warning message. Warnings do not stop the mapping execution.



Icon	Meaning
	They may be generated, for example, when you do not create connections to some mandatory input connectors. In such cases, output will still be generated for those component where valid connections exist.
	Denotes an error. When an error occurs, the mapping execution fails, and no output is generated. The preview of the XSLT or XQuery code is also not possible.

To highlight on the mapping area the component or structure which triggered the information, warning, or error message, click the underlined text in the Messages window.

For components that transform data (such as functions or variables), MapForce validation works as follows:

- If a mandatory **input connector** is unconnected, an error message is generated and the transformation is stopped.
- If an **output connector** is unconnected, then a warning is generated and the transformation process continues. The offending component and its data are ignored and are not mapped to the target document.

To display the result of each validation in an individual tab, click the numbered tabs available on the left side of the Messages window. This may be useful, for example, if you work with multiple mapping files simultaneously

Other buttons in the Messages window enable you to take the following actions:

- Filter the message by types (for example, to show only errors or warnings)
- Move up or down through the entries
- Copy the message text to the clipboard
- Find a specific text in the window
- Clear the Messages window.

For general information about the Messages window, see [User Interface Overview](#).

### 4.1.5 Validating the Mapping Output

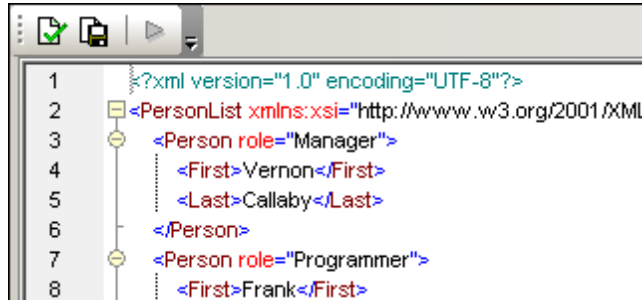
After you click the **Output** tab to preview the mapping, the resulting output becomes available in the Output pane. You can validate this output against the schema associated with it. For example, if the mapping transformation generates an XML file, then the resulting XML document can be validated against the XML schema.

For XML files, you can specify the schema associated with the instance file in the **Add Schema/DTD reference** field of the Component Settings dialog box (see [XML Component Settings](#)). The path specifies where the schema file referenced by the produced XML output is to be located. This ensures that the output instance can be validated when the mapping is executed. You can enter an `http://` address in this field, as well as an absolute or relative path. If you do not select the **Add Schema/DTD reference** field, then the validation of the output file against the schema is not possible. If you select this check box but leave it empty, then the schema filename of the Component Settings dialog box is generated into the output and the validation is done against it.



To validate the mapping output, do one of the following:


- Click the **Validate Output**  toolbar button.



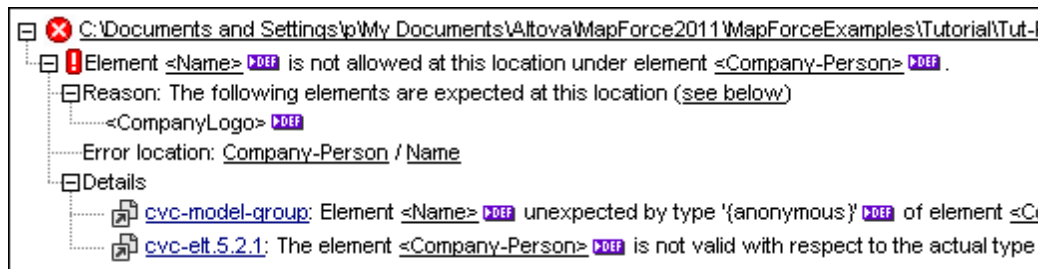
- On the **Output** menu, click **Validate Output File**.

Note: The **Validate Output** button and its corresponding menu command (**Output | Validate Output File**) are enabled only if the output file supports validation against a schema.

The result of the validation is displayed in the Messages window, for example:

 ...Tutorial\ExpReport-Target.xml: Output file validation successful. - 0 error(s), 0 warning(s)

If the validation was not successful, the message contains detailed information on the errors that occurred.



The validation message contains a number of hyperlinks you can click for more detailed information:

- Clicking the file path opens the output of the transformation in the **Output** tab of MapForce.
- Clicking `<ElementName>` link highlights the element in the **Output** tab.
- Clicking the `[DEF]` icon opens the definition of the element in [XMLSpy](#) (if installed).
- Clicking the hyperlinks in the Details subsection (e.g., `cvc-model-group`) opens a description of the corresponding validation rule on the <https://www.w3.org/> website.

## 4.1.6 Previewing the Output

When working with MapForce mappings, you can preview the resulting output without having to run and compile the generated code with an external processor or compiler. In general, it is a good idea to preview the transformation output within MapForce before attempting to process the generated code externally.



When you choose to preview the mapping results, MapForce executes the mapping and populates the Output pane with the resulting output.

Once data is available in the Output pane, you can validate and save it if necessary (see [Validating the Mapping Output](#)). You can also use the **Find** command (**Ctrl + F** key combination) to quickly locate a particular text pattern within the output file (see also [Searching in Text View](#)).

Any errors, warning, or information messages related to the mapping execution are displayed in the Messages window (see [User Interface Overview](#)).

To preview the transformation output:

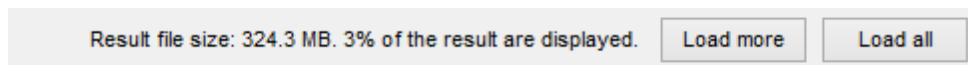
- Click the **Output** tab under the Mapping window. MapForce executes the mapping using the transformation language selected in the Language toolbar and populates the Output pane with the resulting output.

To save the transformation output, do one of the following:

- On the **Output** menu, click **Save Output File**.
- Click the **Save Generated Output** toolbar button.

### Partial output preview

When you are previewing large output files, MapForce limits the amount of data displayed in the Output pane. More specifically, MapForce displays only a part of the file in the Output pane, and a **Load more...** button appears in the lower area of the pane. Clicking the **Load more...** button appends the next file part to the currently visible data, and so on.



**Note:** The **Pretty-print** button becomes active when the complete file has been loaded into the Output pane.

You can configure the preview settings from the **General** tab of the **Options** dialog box (see [Changing the MapForce Options](#)).

## 4.1.7 Text View Features

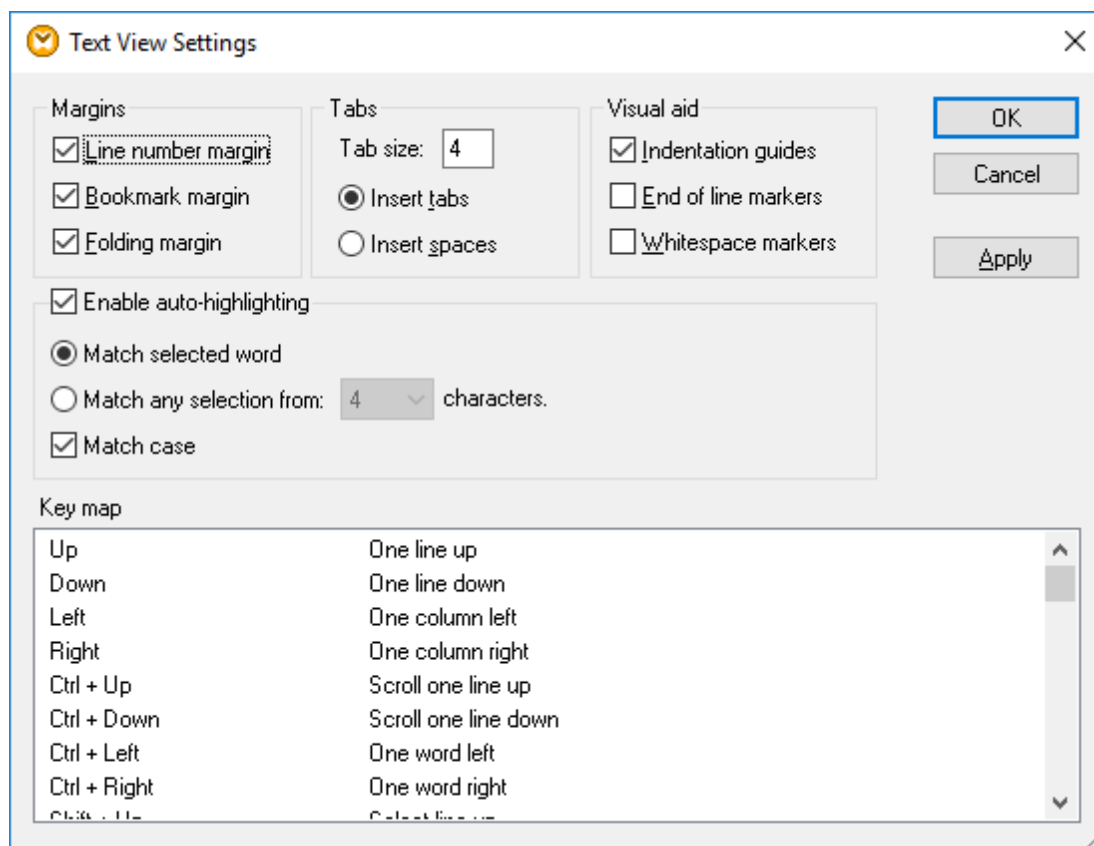
The **Output** pane and the **XSLT** pane have multiple visual aids to make the display of text easier. These include:

- [Line Numbers](#)
- [Syntax Coloring](#)
- [Bookmarks](#)
- [Source Folding](#)
- [Indentation Guides](#)
- [End-of-Line and Whitespace Markers](#)
- [Zooming](#)
- [Pretty-printing](#)




- [Word wrapping](#)
- [Text highlighting](#)

Where applicable, you can toggle or customize the features above from the **Text View Settings** dialog box. Settings in the **Text View Settings** dialog box apply to the entire application—not only to the active document.

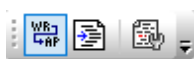


*Text View Settings dialog box*

To open the **Text View settings** dialog box, do one of the following:

- On the **Output** menu, select **Text View Settings**.
- Click the **Text View Settings**  toolbar button.
- Right-click the Output pane, and select **Text View Settings** from the context menu.

Some of the navigation aids can also be toggled from the Text View toolbar, the application menu, or keyboard shortcuts.



*Text View toolbar*

For reference to all applicable shortcuts, see the "Key Map" section of the **Text View Settings** dialog box illustrated above.



### Line numbers

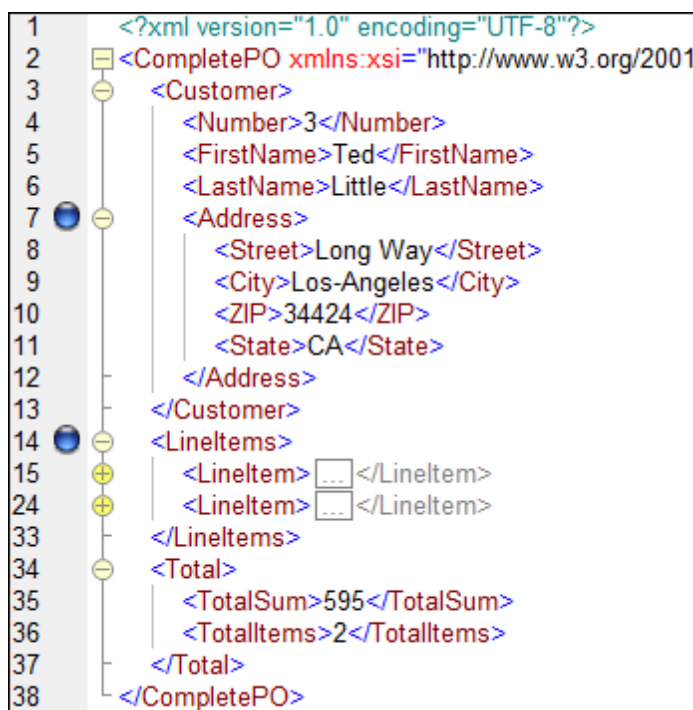
Line numbers are displayed in the line numbers margin, which can be toggled on and off in the **Text View Settings** dialog box. When a section of text is collapsed, the line numbers of the collapsed text are also hidden.

### Syntax coloring

Syntax coloring is applied according to the semantic value of the text. For example, in XML documents, depending on whether the XML node is an element, attribute, content, CDATA section, comment, or processing instruction, the node name (and in some cases the node's content) is colored differently.

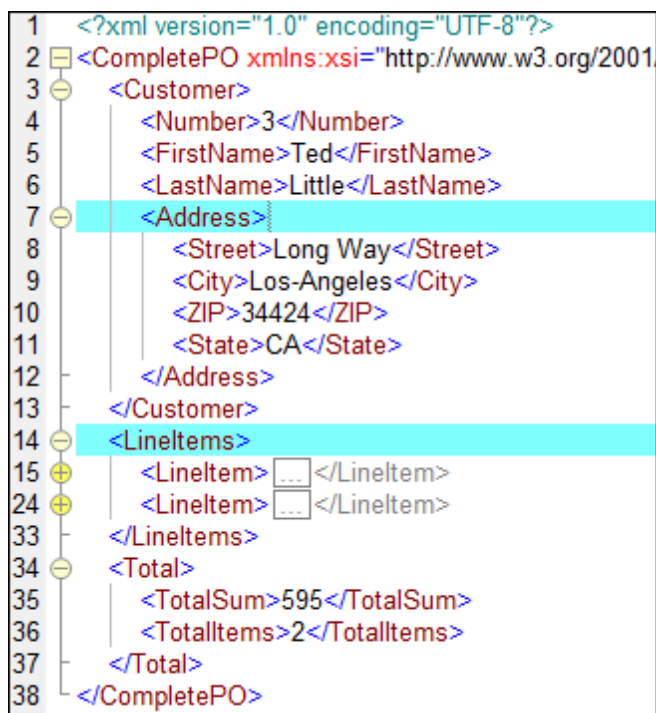
### Bookmarks

Lines in the document can be bookmarked for quick reference and access. If the bookmarks margin is toggled on, bookmarks are displayed in the bookmarks margin.



Otherwise, bookmarked lines are highlighted in cyan.









```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <CompletePO xmlns:xsi="http://www.w3.org/2001.
3  <Customer>
4      <Number>3</Number>
5      <FirstName>Ted</FirstName>
6      <LastName>Little</LastName>
7  <Address>
8      <Street>Long Way</Street>
9      <City>Los-Angeles</City>
10     <ZIP>34424</ZIP>
11     <State>CA</State>
12 </Address>
13 </Customer>
14 <Lineltems>
15     <Lineltem>...</Lineltem>
24     <Lineltem>...</Lineltem>
33 </Lineltems>
34 <Total>
35     <TotalSum>595</TotalSum>
36     <TotalItems>2</TotalItems>
37 </Total>
38 </CompletePO>

```

The bookmarks margin can be toggled on or off in the **Text View Settings** dialog box.

You can edit and navigate bookmarks using the following commands:

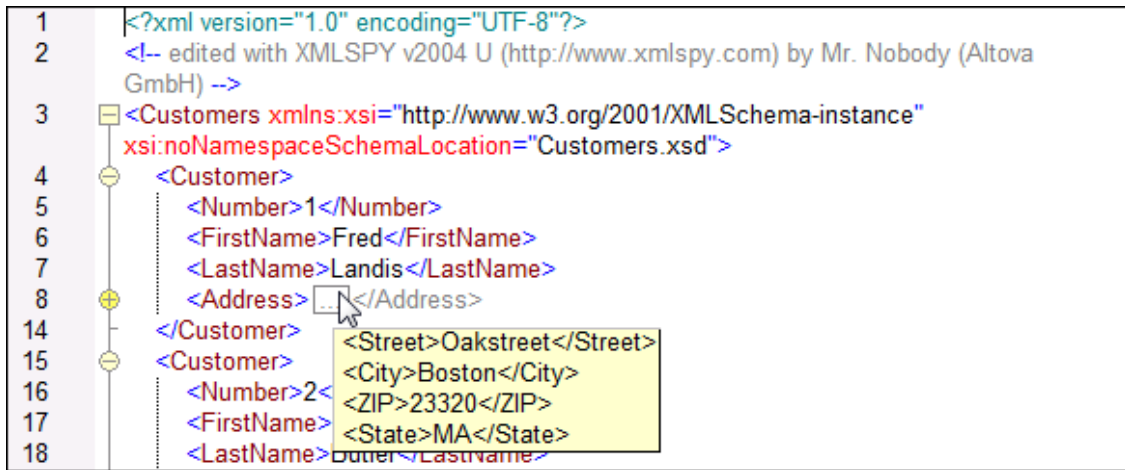
-  **Insert/Remove Bookmark (Ctrl + F2)**
-  **Go to Next Bookmark (F2)**
-  **Go to Previous Bookmark (Shift + F2)**
-  **Delete All Bookmarks (Ctrl + Shift + F2)**

The commands above are available in the **Output** menu. Bookmark commands are also available through the context menu, when you right-click the **Output** (or **XSLT**, or **XQuery**) pane.

### Source folding

Source folding refers to the ability to expand and collapse nodes and is displayed in the source folding margin. The margin can be toggled on and off in the Text View Settings dialog box. To expand or collapse portions of text, click the "+" and "-" nodes at the left side of the window. Any portions of collapsed code are displayed with an ellipsis symbol. To preview the collapsed code without expanding it, move the mouse cursor over the ellipsis. This opens a tooltip that displays the code being previewed, as shown in the image below. Note that, if the previewed text is too big to fit in the tooltip, an additional ellipsis appears at the end of the tooltip.





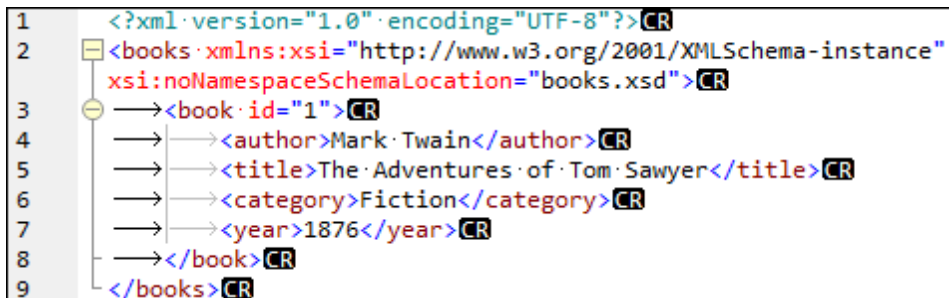
### Indentation guides

Indentation guides are vertical dotted lines that indicate the extent of a line's indentation. They can be toggled on and off in the **Text View Settings** dialog box.

**Note:** The **Insert tabs** and **Insert spaces** options take effect when you use the **Output | Pretty-Print XML text** option.

### End-of-line markers, whitespace markers

End-of-line (EOL) markers and whitespace markers can be toggled on in the **Text View Settings** dialog box. The image below shows a document where both end-of-line and whitespace markers are visible. An arrow represents a tab character, a "CR" is a carriage return, and a dot represents a space character.



### Zooming in and out

You can zoom in and out by scrolling (with the scroll-wheel of the mouse) while holding the **Ctrl** key pressed. Alternatively, press the "-" or "+" keys while holding the **Ctrl** key pressed.


### Pretty-printing

The **Pretty-Print XML Text** command reformats the active XML document in Text View to give a structured display of the document. By default, each child node is offset from its parent by four space characters. This can be customized from the **Text View Settings** dialog box.



To pretty-print an XML document, select the **Output | Pretty-Print XML Text** menu command, or click the **Pretty Print**  toolbar button.

### Word wrapping

To toggle word wrapping in the currently active document, select the **Output | Word Wrap** menu command, or click the **Word Wrap**  toolbar button.

### Text highlighting

When you select text, all matches in the document of the text selection that you make are highlighted automatically. The selection is highlighted in pale blue, and matches are highlighted in pale orange. The selection and its matches are indicated in the scroll bar by gray marker-squares. The current cursor position is given by the blue cursor-marker in the scroll bar.

To switch text highlighting on, select **Enable auto-highlighting** in the Text View Settings dialog box. A selection can be defined to be an entire word or a fixed number of characters. You can also specify whether casing should be taken into account or not.



For a character selection, you can specify the minimum number of characters that must match, starting from the first character in the selection. For example, you can choose to match two or more characters. In this case, one-character selections will not be matched, but a selection consisting of two or more characters will be matched. So, in this case, if you select **t**, then no matches will be shown; selecting **ty** will show all **ty** matches; selecting **typ** will show all **typ** matches; and so on.

For word searches, the following are considered to be separate words: element names (without angular brackets), the angular brackets of element tags, attribute names, and attribute values without quotes.

## 4.1.8 Searching in Text View

The text in the **Output** pane and the **XSLT** pane can be searched using an extensive set of options and visual aids.

To start a search, press **Ctrl+F** (or select the menu command **Edit | Find**). You can then search in the entire document or within a text selection for a search term that you enter in the dialog.




- Enter a string to find, or use the combo box to select a string from one of the last 10 strings.
- When you enter or select a string to find, all matches are highlighted and the positions of the matches are indicated by beige markers in the scroll bar.
- The currently selected match has a different highlight color than the other matches, and its position is indicated in the scroll bar by the dark blue cursor-marker.
- The total number of matches is listed below the search term field, together with the index position of the currently selected match. For example, **2 of 4** indicates that the second of four matches is currently selected.
- You can move from one match to the next, in both directions, by selecting the **Previous**  (**Shift+F3**) and **Next**  (**F3**) buttons at bottom right.






To close the Find dialog, click the **Close**  button at top right, or press **Esc**.

Note the following points:





- The Find dialog is *modeless*. This means that it can remain open while you continue to use Text View.
- If text is selected prior to opening the dialog box, then the selected text is automatically inserted into the search term field.
- To search within a selection, do the following: (i) Mark the selection; (ii) Toggle on the *Find in Selection*  option to lock the selection; (iii) Enter the search term. To search within another selection, unlock the current selection by toggling off the *Find in Selection*  option, then make the new selection and toggle on the *Find in Selection*  option.
- After the Find dialog is closed, you can repeat the current search by pressing **F3** for a forward search, or **Shift+F3** for a backward search. The Find dialog will appear again in this case.

**Find options**


Find criteria can be specified via buttons located below the search term field. When an option is toggled on, its button color changes to blue. You can select from the following options:


Option	Icon	Description
Match case		Performs a case-sensitive search when toggled on ("Address" is not the same as "address").



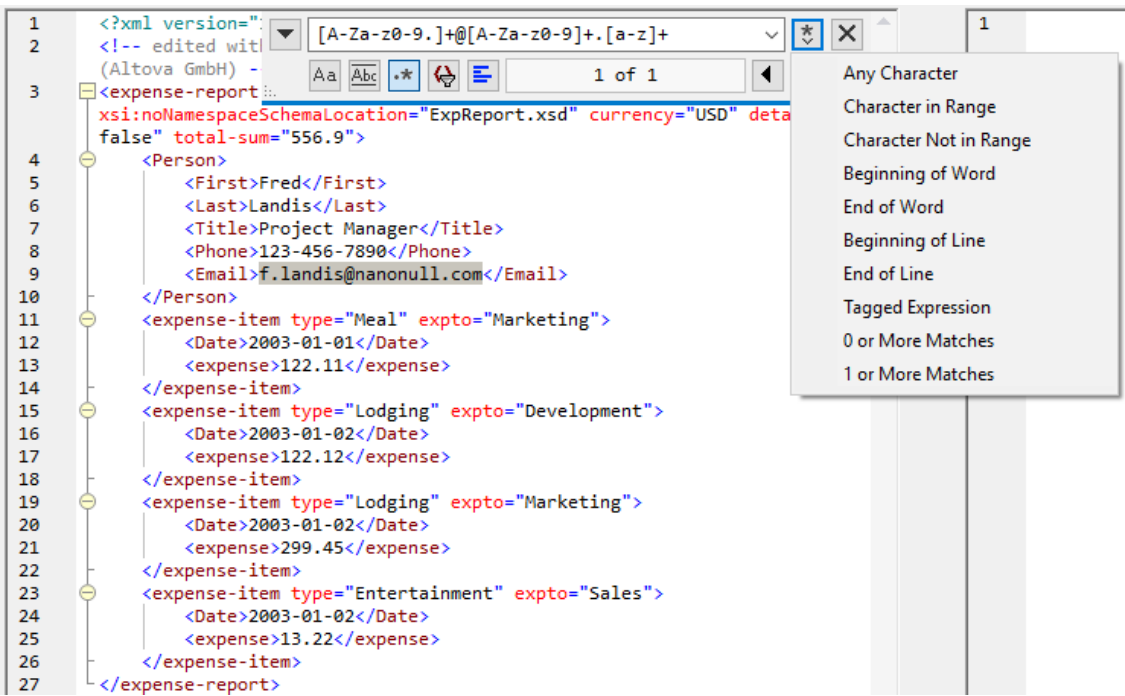
Option	Icon	Description
Match whole word		Only the exact words in the text will be matched. For example, for the input string <i>fit</i> , with <b>Match whole word</b> toggled on, only the word <i>fit</i> will match the search string; the <i>fit</i> in <i>fitness</i> , for example, will not.
Regular expression		If toggled on, the search term will be read as a regular expression. See "Using regular expressions" below.
Find anchor		When a search term is entered, the matches in the document are highlighted and one of these matches will be marked as the current selection. The <b>Find anchor</b> toggle determines whether that first current selection is made relative to the cursor position or not. If <b>Find anchor</b> is toggled on, then the first currently selected match will be the next match from the current cursor location. If <b>Find anchor</b> is toggled off, then the first currently selected match will be the first match in the document, starting from the top.
Find in selection		When toggled on, locks the current text selection and restricts the search to the selection. Otherwise, the entire document is searched. Before selecting a new range of text, unlock the current selection by toggling off the Find in Selection option.

### Using regular expressions

You can use regular expressions (regex) to find a text string. To do this, first, switch the *Regular expression*  option on. This specifies that the text in the search term field is to be evaluated as a regular expression. Next, enter the regular expression in the search term field. For help with

building a regular expression, click the **Regular Expression Builder**  button, which is located to the right of the search term field. Click an item in the Builder to enter the corresponding regex metacharacter/s in the search term field. The screenshot below shows a simple regular expression to find email addresses.





The following custom set of regular expression metacharacters are supported when finding and replacing text.

.	Matches any character. This is a placeholder for a single character.
( abc )	<p>The ( and ) metacharacters mark the start and end of a tagged expression. Tagged expressions may be useful when you need to tag ("remember") a matched region for the purpose of referring to it later (back-reference). Tagged expressions are similar to matched subexpressions (indexed groups) in the .NET flavour of regular expressions. Up to nine sub-expressions can be tagged (and then back-referenced later).</p> <p>For example, <b>(the) \1</b> matches the string <code>the the</code>. This expression can be literally explained as follows: match the string "the" (and remember it as a tagged region), followed by a space character, followed by a back-reference to the tagged region matched previously.</p>
\n	Where <i>n</i> is 1 through 9 , <i>n</i> refers to the first through ninth tagged region (see above).
\<	Matches the start of a word.
\>	Matches the end of a word.
\	Escapes the character following the backslash. In other words, the expression <code>\x</code> allows you to use the character <i>x</i> literally. For example, <code>\[</code> would be interpreted as <code>[</code> and not as the start of a character set.
[ ... ]	Matches any characters in this set. For example, <b>[abc]</b> matches any of the characters <code>a</code> , <code>b</code> or <code>c</code> . You can also use ranges: for example <b>[a-z]</b> for any lower case character.



[^...]	Matches any characters not in this set. For example, <code>[^A-Za-z]</code> matches any character except an alphabetic character.
^	Matches the start of a line (unless used inside a set, <i>see above</i> ).
\$	Matches the end of a line. For example, <code>A+\$</code> matches one or more A's at end of line.
*	Matches zero or more occurrences of the preceding expression. For example, <code>Sa*m</code> matches <code>Sm</code> , <code>Sam</code> , <code>Saam</code> , <code>Saaam</code> and so on.
+	Matches one or more occurrences of the preceding expression. For example, <code>Sa+m</code> matches <code>Sam</code> , <code>Saam</code> , <code>Saaam</code> and so on.

### Finding special characters

You can search for any the following special characters within text, provided that the **Regular**

**expression** option  is enabled:

- `\t` (Tab)
- `\r` (Carriage Return)
- `\n` (New line)
- `\\` (Backslash)

For example, to find a tab character, press **Ctrl + F**, select the  option, and then enter `\t` in the Find dialog box.

## 4.1.9 Previewing the XSLT Code

You can preview the XSLT code generated by MapForce if you selected XSLT 1.0 or XSLT 2.0 as data transformation language (see [Selecting a transformation language](#)).

To preview the generated XSLT 1.0 (or XSLT 2.0) code, do one of the following:

- To preview the XSLT 1.0 code, click the **XSLT** tab under the Mapping window.
- To preview the XSLT 2.0 code, click the **XSLT2** tab under the Mapping window.

**Note:** The XSLT (or XSLT2) tab becomes available if you have selected XSLT (or XSLT2, respectively) as transformation language.

## 4.1.10 Generating XSLT Code

To generate XSLT code:

1. Select the menu item **File | Generate code in | XSLT 1.0 (XSLT 2.0)**.
2. Select the folder you want to save the generated XSLT file, and click **OK**. MapForce generates the code and displays the result of the operation in the Messages window.

The name of the generated .xslt file has the form **<A>MapTo<B>.xslt**, where:

- "<A>" is the value of the **Application Name** field in mapping settings (see [Changing the Mapping Settings](#)).



- "<B>" is the name of the target mapping component. To change this value, open the settings of the target component and edit the value of the **Component Name** field (see [Changing the Component Settings](#)).

The folder where the .xslt file is saved also contains a batch file called **DoTransform.bat** which can be run with RaptorXML Server to transform the data (see [Automation with RaptorXML Server](#)).

**To run the transformation with RaptorXML Server:**

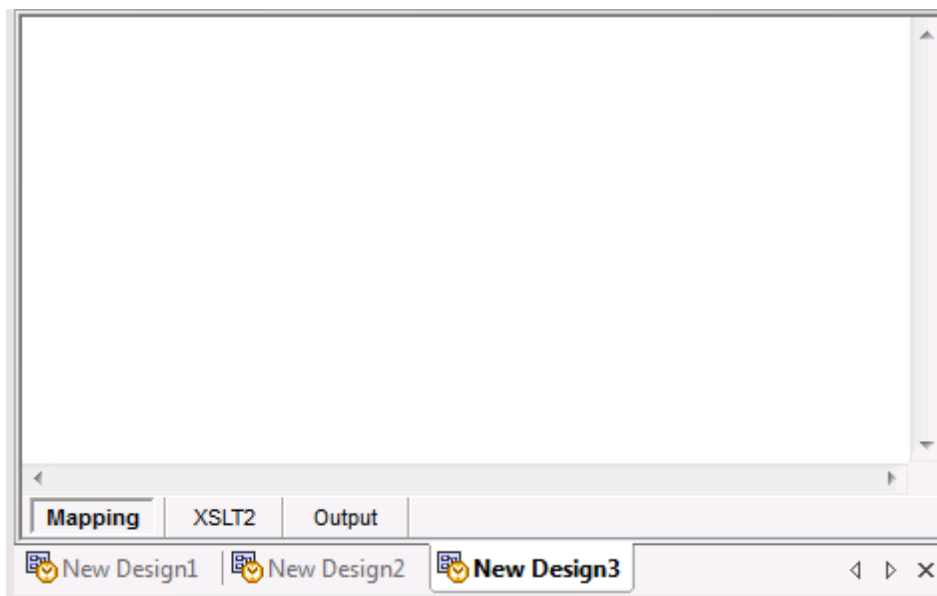
1. Download and install RaptorXML from the download page (<https://www.altova.com/download#server>).
2. Start the **DoTransform.bat** batch file located in the previously designated output folder.

Note that you might need to add the RaptorXML installation location to the **path** variable of the Environment Variables. You can find the RaptorXML documentation on the website documentation page (<https://www.altova.com/documentation>).

## 4.1.11 Working with Multiple Mapping Windows

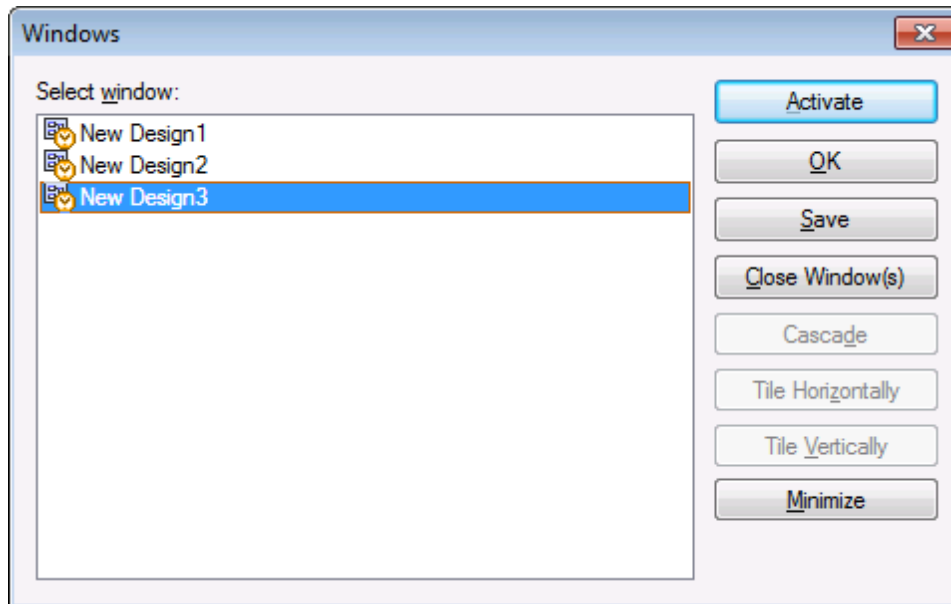
MapForce uses a Multiple Document Interface (MDI). Each mapping file you open in MapForce has a separate window. This enables you to work with multiple mapping windows and arrange or resize them in various ways inside the main (parent) MapForce window. You can also arrange all open windows using the standard Windows layouts: Tile horizontally, Tile vertically, Cascade.

When multiple mappings are open in MapForce, you can quickly switch between them using the tabs displayed in the lower part of the Mapping pane.



Window management options are available both on the **Window** menu and on the **Windows** dialog box. From the **Windows** dialog box, you can take actions against any or all currently open mapping windows (including saving, closing, or minimizing them).





*Windows dialog box*

You can open the Windows dialog box using the menu command **Window | Windows...** To select multiple windows in the Windows dialog box, click the required entries while holding the **Ctrl** key pressed.

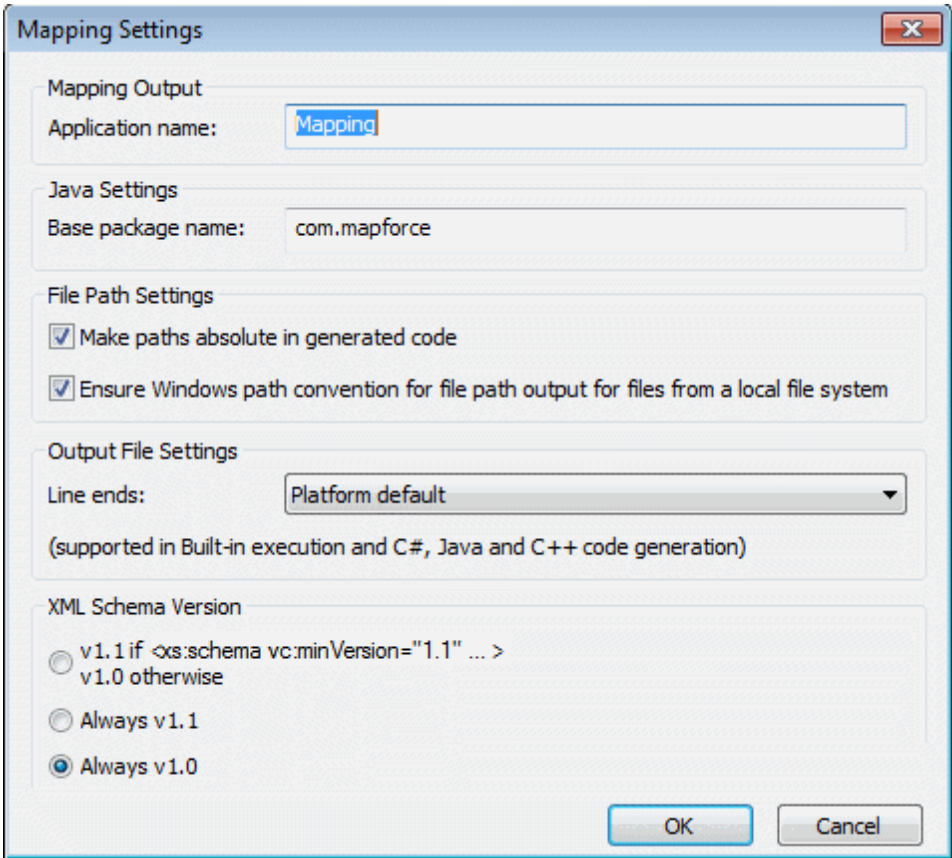
#### 4.1.12 Changing the Mapping Settings

You can change the document-specific settings of the currently active mapping design file from the Mapping Settings dialog box. This information is stored in the \*.mfd file.

**To open the Mapping Settings dialog box:**

- On the **File** menu, click **Mapping Settings**.





Mapping Settings dialog box

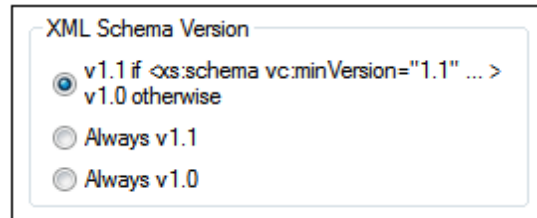
The available settings are as follows.

<i>Application Name</i>	Defines the XSLT1.0/2.0 file name prefix for the generated transformation files.
<i>Make paths absolute in generated code</i>	Defines whether the file paths should be relative or absolute in the generated program code. For more information, see <a href="#">About Paths in Generated Code</a> .
<i>Ensure Windows path convention for file path</i>	<p>The "Ensure Windows path convention...." check box makes sure that Windows path conventions are followed. When outputting XSLT2 (and XQuery), the currently processed file name is internally retrieved using the document-uri function, which returns a path in the form file:// URI for local files.</p> <p>When this check box is active, a file:// URI path specification is automatically converted to a complete Windows file path (e.g. "C:\...") to simplify further processing.</p>
<i>XML Schema Version</i>	Lets you define the XML Schema Version used in the mapping file. You can define if you always want to <b>load</b> the Schemas conforming to version 1.0 or 1.1. Note that not all



version 1.1 specific features are currently supported.

If the `xs:schema vc:minVersion="1.1"` declaration is present, then version 1.1 will be used; if not, version 1.0 will be used.



If the XSD document has no `vc:minVersion` attribute or the value of the `vc:minVersion` attribute is other than 1.0 or 1.1, then XSD 1.0 will be the default mode.

**Note:** Do not confuse the `vc:minVersion` attribute with the `xsd:version` attribute. The former holds the XSD version number, while the latter holds the document version number.

Changing this setting in an existing mapping causes a reloading of all schemas of the selected XML schema version, and might also change its validity.



## 4.2 Working with Components

Components are the central elements of any mapping design in MapForce. Generally, the term "component" is a convenient way to call any object which acts as a data source, or as a data target, or represents your data in the mapping at an intermediary processing stage.

There are two main categories of components: structure components and transformation components.

The structure components represent the abstract structure or schema of your data. For example, when you add an XML file to the mapping area (using the menu command **Insert | XML Schema/ File**), it becomes a mapping component. For further information about structure components and their specifics, see [Data Sources and Targets](#). With a few exceptions, structure components consist of items and sequences. An item is the lowest level mapping unit (for example, a single attribute in the XML file, or an element of simple type). A sequence is a collection of items.

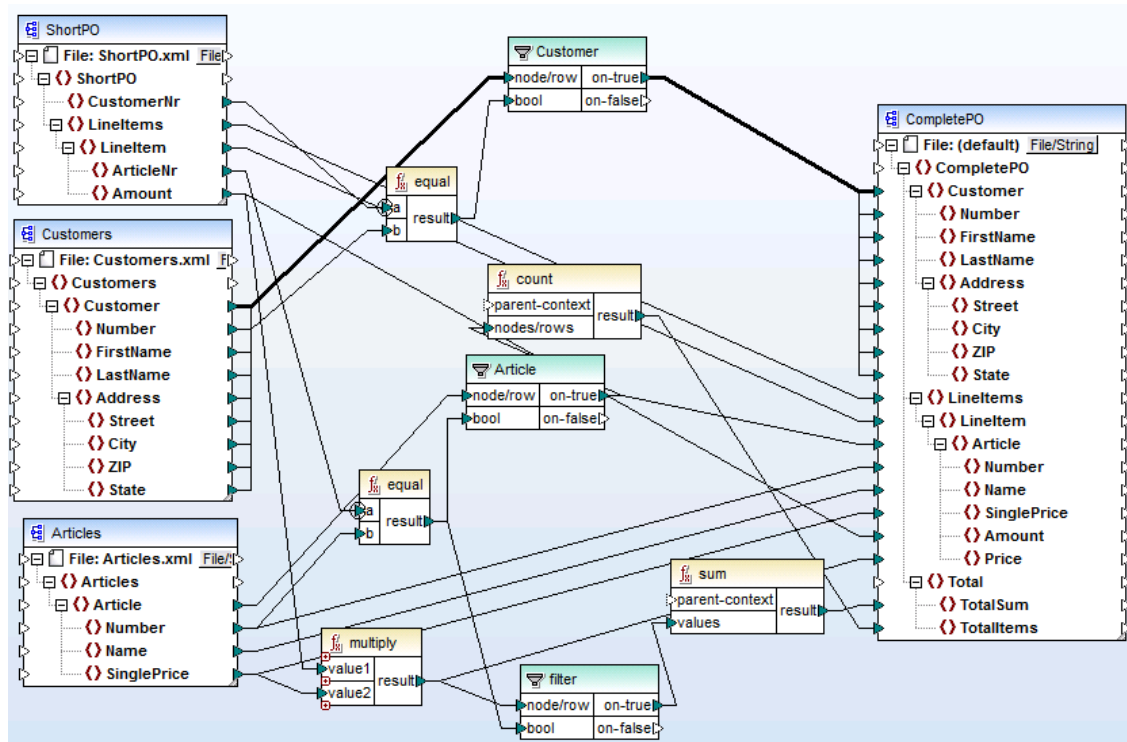
The transformation components either transform data (for example, functions), or assist you in transformations (for example, constants or variables). For information on how you can use these components to achieve various data transformation tasks, see [Designing Mappings](#).

With the help of structure components, you can either read data from files or other sources, write data to files or other sources, or store data at some intermediary stage in the mapping process (for example, in order to preview it). Consequently, structure components can be of the following types:

- **Source.** You declare a component as source by placing it on the left of the mapping area, and, thus, instructing MapForce to read data from it.
- **Target.** You declare a component as target by placing on the right of the mapping area, and, thus, instructing MapForce to write data to it.
- **Pass-through.** This is a special component type which acts both as a source and target (for further information, see [Chained mappings / pass-through components](#)).

On the mapping area, components appear as rectangles. The following sample mapping illustrates three source components, one target XML component, and various transformation components (functions and filters) through which data goes before being written to the source.





CompletePO.mfd

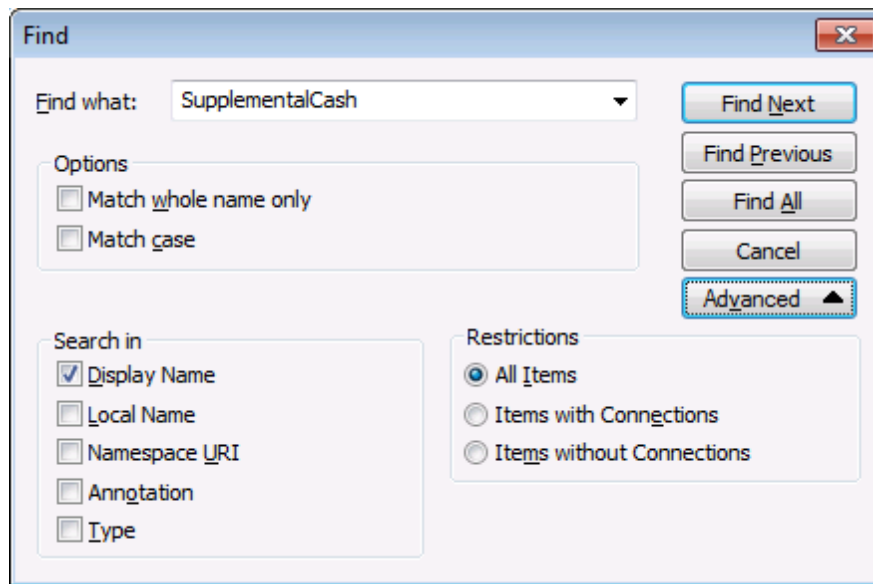
This mapping sample is available at the following path: <Documents>\Altova\MapForce2018\MapForceExamples\CompletePO.mfd.

### 4.2.1 Searching within Components

To search for a specific node/item in a component:

1. Click the component you want to search in, and press the CTRL+F keys.
2. Enter the search term and click **Find Next**.



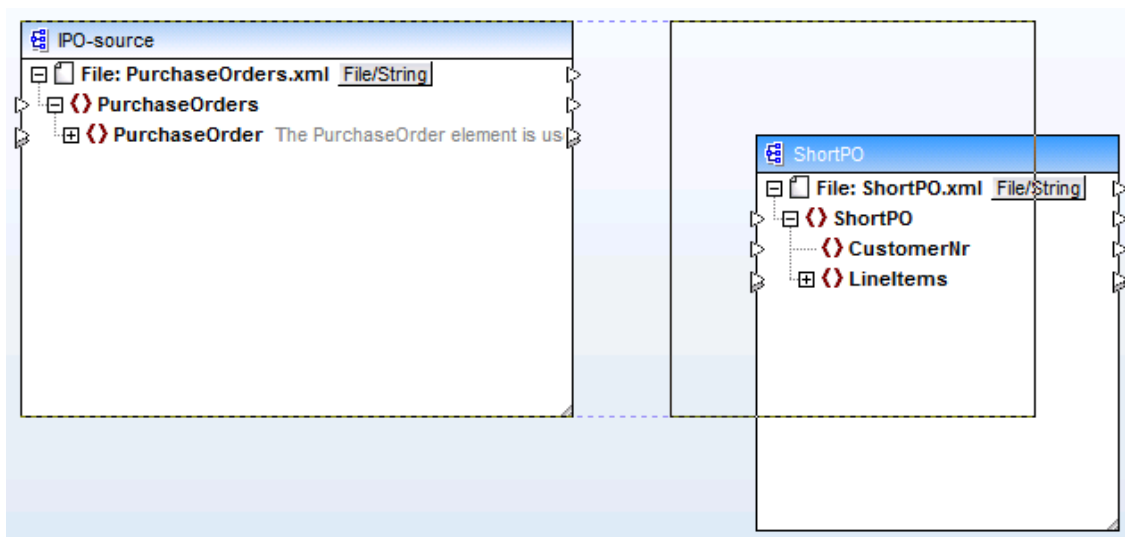


Use the Advanced options to define which items (nodes) are to be searched, as well as restrict the search options based on the specific connections.

## 4.2.2 Aligning Components

When you move components in the mapping pane, MapForce displays auto-alignment guide lines. These guide lines help you align a component to any other component in the mapping window.

In the sample mapping below, the lower component is being moved. The guide lines show that it can be aligned to the component on the left side of the mapping.



Component auto-alignment guide lines



**To enable or disable this option:**


1. On the **Tools** menu, click **Options**.
2. In the **Editing** group, select the **Align components on mouse dragging** check box.

### 4.2.3 Changing the Component Settings

After you add a component to the mapping area, you can configure the settings applicable to it from the Component Settings dialog box. You can open the Component settings dialog box in one of the following ways:

- Select the component and, on the **Component** menu, click **Properties**.
- Double-click the component header.
- Right-click the component header, and then click **Properties**.

For a description of the settings available on the Component Settings dialog box, see [XML Component Settings](#).

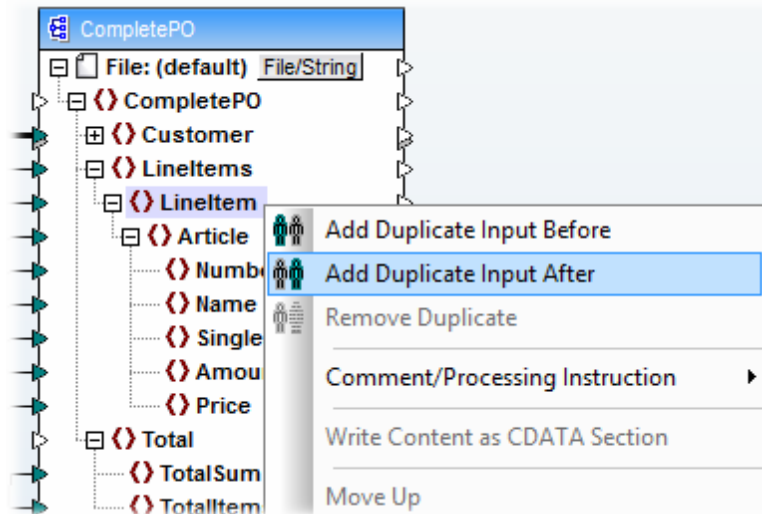
For any file-based component, such as XML, a **File** (  ) button appears next to the root node. This button specifies advanced options applicable if you want to process or generate multiple files in a single mapping (see [Processing Multiple Input or Output Files Dynamically](#)).

### 4.2.4 Duplicating Input

Sometimes, you may need to configure a component to accept data from more than one source. For example, you may need to convert data from two different XML schemas into a single schema. To make the destination schema accept data from both source schemas, you can duplicate any of the input items in the component. Duplicating input is meaningful only for a component which is a target component. On any given target component, you can duplicate as many items as required.

To duplicate a particular input item, right-click it and select **Add Duplicate Input After/Before** from the context menu.





In the image above, the item `LineItem` is being duplicated in order to provide the ability to map data from a second source.

Once you duplicate an input, you can make connections both to the original input and to the duplicate input. For example, this would enable you to copy data from source A to original input, and data from source B to the duplicate input.

**Note:** Duplication of XML attributes is not allowed, as it would make the resulting XML instance invalid. In case of XML elements, duplicating input is allowed regardless of the value of the element's `maxOccurs` attribute in the schema. This behaviour is intentional, since the schema could change later, or the source data could be optional. For example, a mapping could generate a single XML element, even if the input is duplicated on the mapping.

For a step-by-step example, see [Map Multiple Sources to One Target](#).



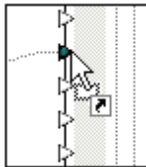
## 4.3 Working with Connections

A mapping is ultimately about transforming data from one format or structure into another. In a very basic mapping scenario, you add to the mapping area the components which represent your source and your target data (for example, a source XML schema and a destination one), and then draw visually the mapping connections between the two structure. A connection is, therefore, the visual representation of how data is mapped from a source to a destination.

Components have inputs and outputs which appear on the mapping as small triangles, called connectors. Input connectors are positioned to the left of any item to which you can draw a connection. Output connectors are positioned to the right of any item from which you can draw a connection.

### To draw a connection between two items:

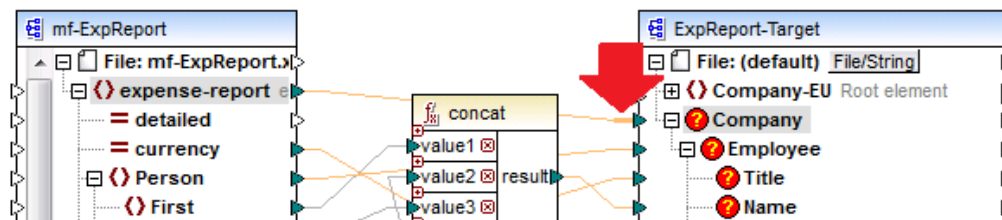
- Click the output connector of a source item and drag it to a destination item. When the drop action is allowed, a link tooltip appears next to the text cursor.



An input connector accepts only one incoming connection. If you try to add a second connection to the same input, a message box appears asking if you want to replace the connection with a new one or duplicate the input item. An output connector can have several connections, each to a different input.

### To move a connection to a different item:

- Click the stub of the connection (the straight section closer to the target) and drag it to the destination.



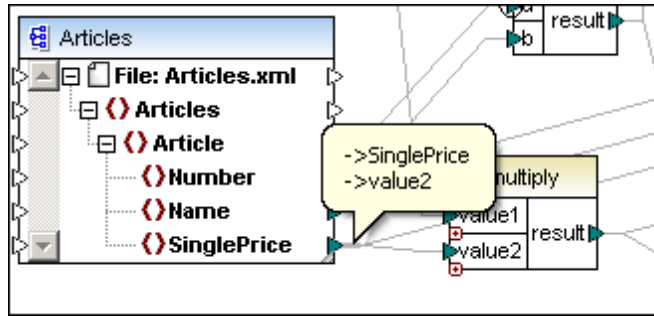
### To copy a connection to a different item:

- Click the stub of the connection (the straight section closer to the target), and drag it to the destination while holding down the **Ctrl** key.



### To view the item(s) at the other end of a connection:

- Point to the straight section of a connection (close to the input/output connector). A tooltip appears which displays the name(s) of the item(s) at the other end of the connection. If multiple connections have been defined from the same output, then a maximum of ten item names are displayed. In the sample below, the two target items are **SinglePrice** and **value2** of the multiply function.



### To change the connection settings, do one of the following:

- On the **Connection** menu, click **Properties** (this menu item becomes enabled when you select a connection).
- Double-click the connection.
- Right-click the connection, and then click **Properties**.

See also [Connection Settings](#).

### To delete a connection, do one of the following:

- Click the connection, and then press the **Delete** key.
- Right-click the connection, and then click **Delete**.

## 4.3.1 About Mandatory Inputs

To aid you in the mapping process, MapForce highlights in orange the mandatory inputs in target components:

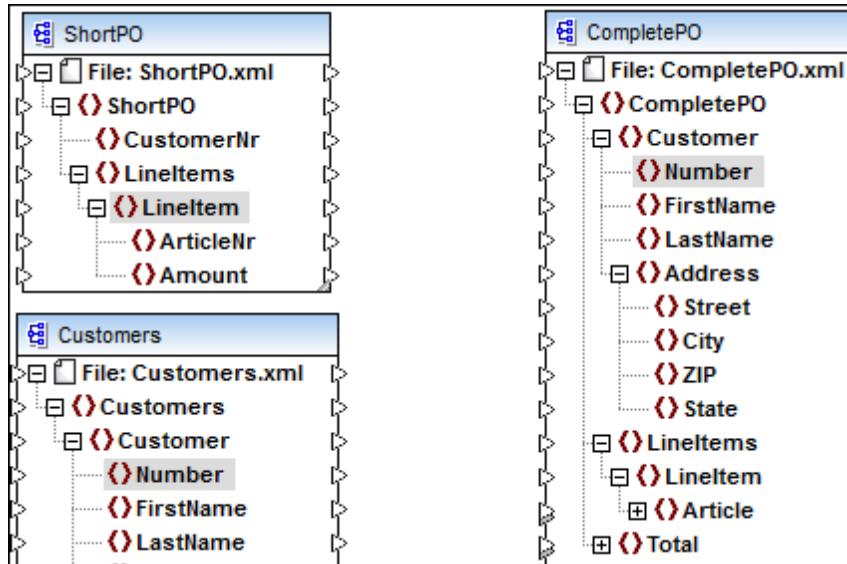
- In XML and EDI components these are items where the minOccurs parameter is equal/greater than 1.
- In databases these are fields that have been defined as "not null"
- WSDL calls and WSDL response (all nodes)
- XBRL nodes that have been defined as mandatory
- In functions these are the specific mandatory parameters such that once one parameter has been mapped, then the other mandatory ones will be highlighted to show that a connection is needed. E.g. once one of the filter input parameters is mapped, then the other one is automatically highlighted.
- Worksheet names in MS Excel sheets

Example:

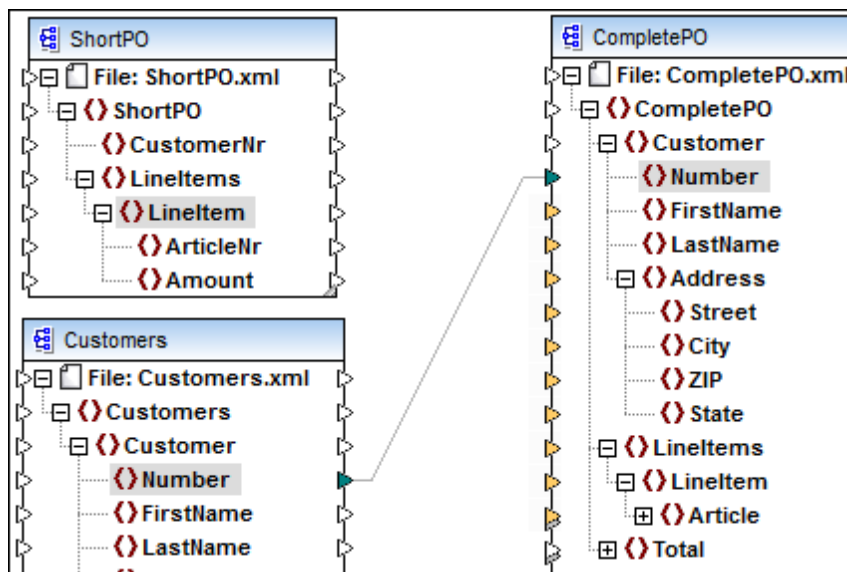
When creating a mapping like CompletePO.mfd, available in the ...\\MapForceExamples folder, the



inserted XML Schema files exist as shown below.



The Number element of the Customers component is then connected to the Number element of the CompletePO component. As soon as the connection has been made, the mandatory items/nodes of the CompletePO component are highlighted. Note that the collapsed "Article" node/icon is also highlighted.



### 4.3.2 Changing the Connection Display Preferences

You can selectively view the connections in the mapping window.



**Show selected component connectors** switches between showing:

- all mapping connectors in black, or
- those connectors relating to the currently selected component in black. Other connectors appear dimmed.






**Show connectors from source to target** switches between showing:

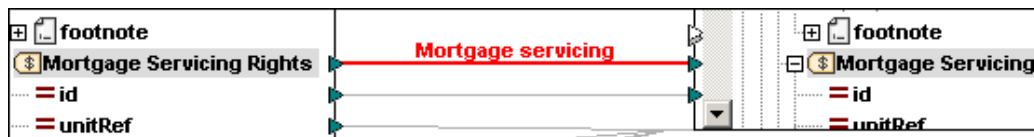
- connectors that are **directly** connected to the currently selected component, or
- connectors linked to the currently selected component, originating from source and terminating at the target components.


### 4.3.3 Annotating Connections

Individual connections can be labeled allowing you to comment your mapping in great detail. This option is available for **all connection types**.

**To annotate to a connection:**

1. Right-click the connection, and select **Properties** from the context menu.
2. Enter the name of the currently selected connection in the **Description** field. This enables all the options in the Annotation Settings group.
2. Use the remaining groups to define the **starting location**, **alignment** and **position** of the label.
3. Activate the **Show annotations**  icon in the View Options toolbar to see the annotation text.

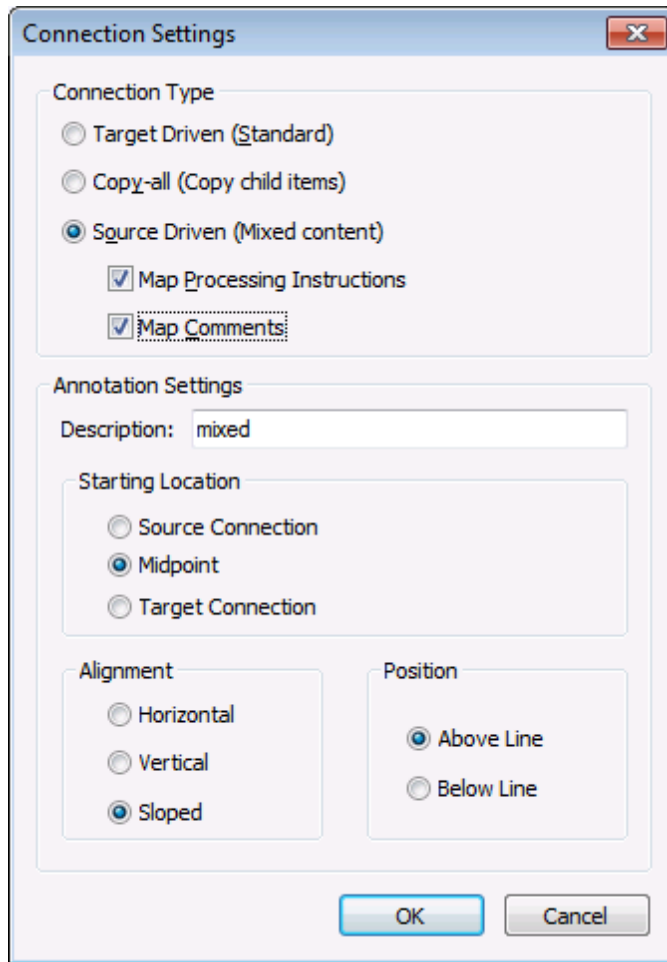


**Note:** If the **Show annotations** icon is inactive, you can still see the annotation text if you place the mouse cursor over the connection. The annotation text will appear in a callout if the **Show tips**  toolbar button is active in the View Options toolbar.

### 4.3.4 Connection Settings

Right-clicking a connection and selecting **Properties** from the context menu, or double-clicking a connection, opens the Connection Settings dialog box in which you can define the settings of the current connection. Note that unavailable options are disabled.





Connection Settings dialog box

For items of **complexType**, you can choose one of the following connection types for mapping (note that these settings also apply to **complexType** items which do not have any text nodes):

<i>Target Driven (Standard)</i>	Changes the connection type to "Target-driven" (see <a href="#">Target-driven / Standard mapping</a> ).
<i>Copy-all (Copy child items)</i>	Changes the connection type to "Copy-all" and automatically connects all identical items in the source and target components (see <a href="#">Copy-all connections</a> ).
<i>Source Driven (mixed content)</i>	Changes the connection type to "Source-driven", and enables the selection of additional elements to be mapped. The additional elements must be child items of the mapped item in the XML source file, to qualify for mapping.  Activating the <b>Map Processing Instructions</b> or <b>Map Comments</b> check boxes enables you to include these data groups in the output file.



6

7

8

9

10

11

<Desc>

<para>The company was established in<b> Vereno</b>in 1995. Nanonull devel<br><i>multi-core processors.</i>February 1999 saw the unveiling of the first prototype <b>h<br>hopes to expand its operations <i>offshore</i>to drive down operational costs.</b><br><?sort alpha-ascending?><br><!--Company details: location and general company information.--><br></para><br><para>White papers and further information will be made available in the near future.</para>

Note: CDATA sections are treated as text.

The Annotation Settings group enables you to annotate the connection (see [Annotating Connections](#) ).

4.3.5 Connection Context Menu

When you right-click a connection, the following context commands are available.

×

Connect Matching Children...

×

Delete

Delete

Go to source: book

Go to target: publication

✓

Target Driven (Standard)

Copy-All (Copy Child Items)

Source Driven (Mixed Content)

A-Z

Insert Sort: Nodes/Rows

Filter

Insert Filter: Nodes/Rows

SQL

Insert SQL-WHERE/ORDER

Value-Map

Insert Value-Map

Properties

Connect matching children	Opens the "Connect Matching Children" dialog box (see <a href="#">Connecting Matching Children</a> ). This command is enabled when the connection is eligible to have matching children.
Delete	Deletes the selected connection.
Go to source: <item name>	Selects the source connector of the current connection.
Go to target: <item name>	Selects the target connector of the current connection.
Target Driven (Standard)	Changes the connection type to "Target-driven" (see <a href="#">Target-driven connections</a> ).



<i>Copy-All (Copy Child Items)</i>	Changes the connection type to "Copy-all" and automatically connects all identical items in the source and target components (see <a href="#">Copy-all connections</a> ).  This command is enabled (and meaningful) when both the source item and the target item have children items.
<i>Source Driven (Mixed Content)</i>	Changes the connection type to "Source-driven" (see <a href="#">Source-driven connections</a> ).  This command is enabled (and meaningful) when both the source item and the target item have children items.
<i>Insert Sort: Nodes/Rows</i>	Adds a Sort component between the source and the target item (see <a href="#">Sorting Data</a> ).
<i>Insert Filter: Nodes/Rows</i>	Adds a Filter component between the source and the target item (see <a href="#">Filters and Conditions</a> ).
<i>Insert Value-Map</i>	Adds a Value-Map component between the source and the target item (see <a href="#">Using Value-Maps</a> ).
<i>Properties</i>	Opens the Connections Settings dialog box (see <a href="#">Connection Settings</a> ).

### 4.3.6 Connecting Matching Children

You can create multiple connections between items of the **same name** in both the source and target components. Note that a "Copy-all" connection (see [Copy-all connections](#)) is created by default.

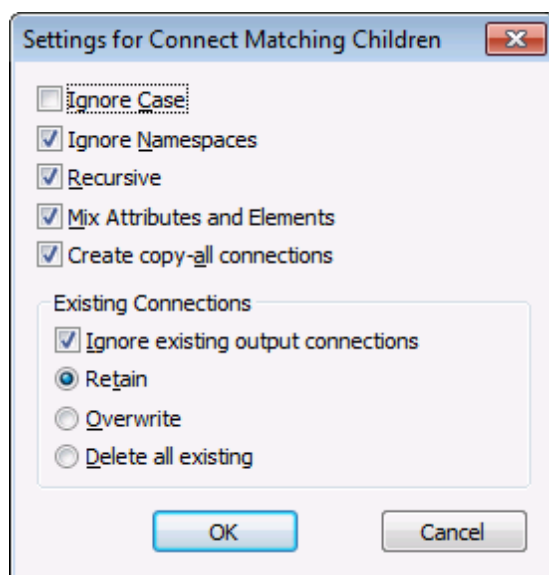
**To toggle the "Auto Connect Matching Children" option on or off, do one of the following:**

- Click the **Auto Connect Matching Children** (  ) toolbar button.
- On the **Connection** menu, click **Auto Connect Matching Children**.


**To change the settings for "Connect Matching Children":**

1. Connect two (parent) items that share identically named **child items** in both components.
2. Right click the connection and select the **Connect matching child elements** option.





3. Select the required options (see the table below), and click OK. Connections are created for all the child items that have identical names and adhere to the settings defined in the dialog box.

**Note:** The settings you define here are applied when connecting two items if the **Toggle auto connect of children** (  ) toolbar button is active.

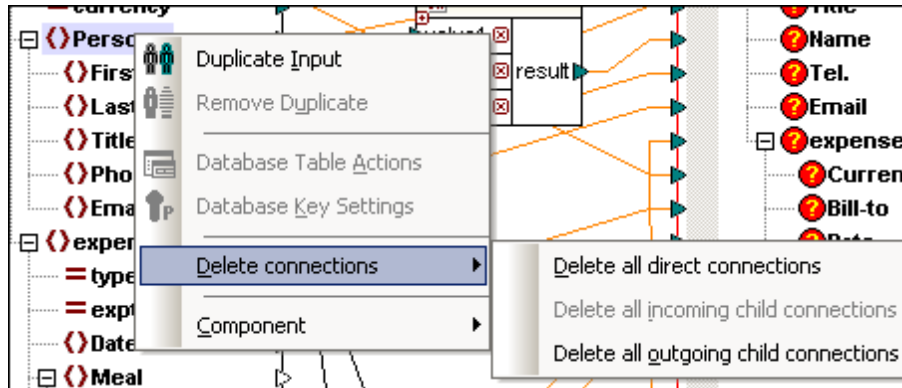
<i>Ignore Case</i>	Ignores the case of the child item names.
<i>Ignore Namespaces</i>	Ignores the namespaces of the child items.
<i>Recursive</i>	Creates new connections between any matching items recursively. That is, a connection is created no matter how deep the items are nested in the hierarchy, as long as they have the same name.
<i>Mix Attributes and Elements</i>	When enabled, allows connections to be created between attributes and elements which have the same name. For example, a connection is created if two "Name" items exist, even though one is an element, and the other is an attribute.
<i>Create copy-all connections</i>	This setting is active by default. It creates (if possible) a connection of type "Copy-all" between source and target items.
<i>Ignore existing output connections</i>	Creates additional connections for any matching items, even if they already have outgoing connections.
<i>Retain</i>	Retains existing connections.
<i>Overwrite</i>	Recreates connections according to the settings defined. Existing connections are discarded.



<i>Delete all existing</i>	Deletes all existing connections, before creating new ones.
----------------------------	-------------------------------------------------------------

### Deleting connections

Connections that have been created using the Connect Matching Children dialog, or during the mapping process, can be removed as a group.



#### To delete connections:

1. Right-click the item name in the component, not the connection itself ("Person" in this example).
2. Select **Delete Connections | Delete all ... connections**.

<i>Delete all direct connections</i>	Deletes all connections directly mapped to, or from, the current component to any other source or target components.
<i>Delete all incoming child connections</i>	Only active if you have right clicked an item in a target component. Deletes all incoming child connections.
<i>Delete all outgoing child connections</i>	Only active if you have right clicked an item in a source component. Deletes all outgoing child connections.

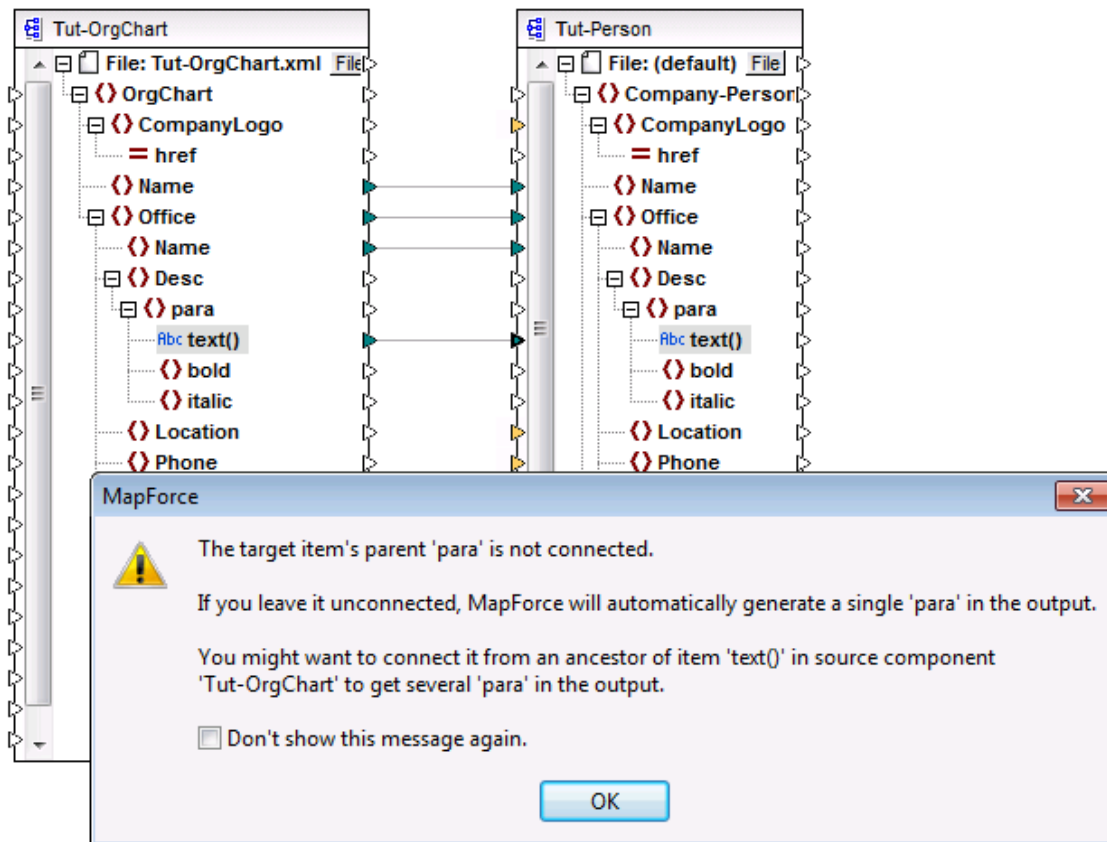
## 4.3.7 Notifications on Missing Parent Connections

When you create connections between source and target items manually, MapForce automatically analyzes the possible mapping outcomes. If you are mapping two child items, a notification message can appear suggesting that you also connect the parent of the source item with the parent in the target item.

This notification message helps you prevent situations where a single child item appears in the Output window when you preview the mapping. This will generally be the case if the source node supplies a sequence instead of a single value.

To understand how this works, open the sample mapping **Tut-OrgChart.mfd** available in the **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\** folder. If you connect the source `text()` item to the target `text()` item, a notification message appears, stating that the parent item "para" is not connected and will only be generated once in the output.





*Tut-OrgChart.mfd (MapForce Basic Edition)*

To generate multiple `para` items in the target, connect the source and target `para` items to each other.

To disable such notifications, do the following:

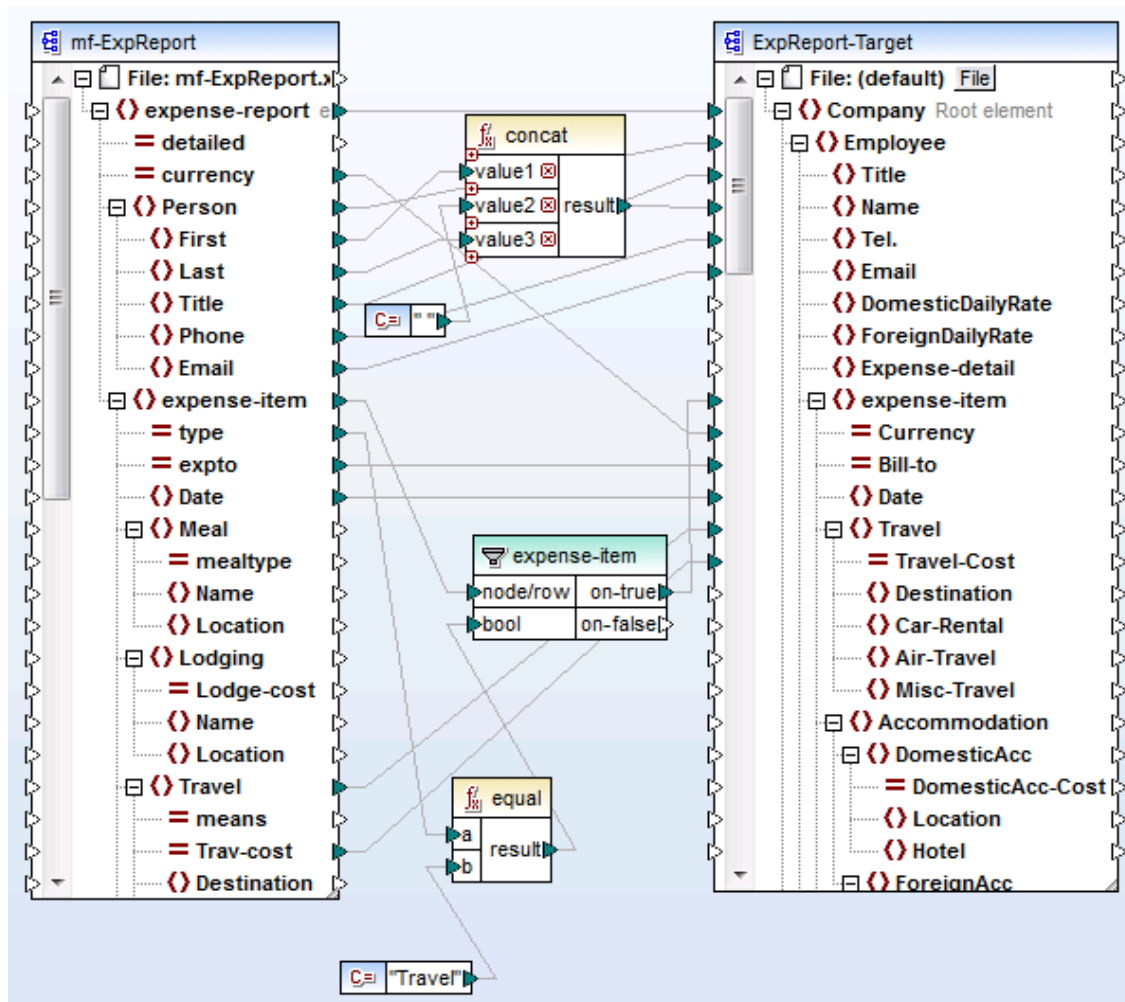
1. On the **Tools** menu, click **Options**.
2. Click the **Messages** group.
3. Click to clear the **When creating a connection, suggest connecting ancestor items** check box.

### 4.3.8 Moving Connections and Child Connections

When you move a connection to a different component, MapForce automatically matches identical child connections and will prompt you whether it should move them to the new location as well. A common use of this feature is if you have an existing mapping and then change the root element of the target schema. Normally, when this happens, you would need to remap all descending connections manually. This feature helps you prevent such situations.

This example uses the **Tut-ExpReport.mfd** file available in the **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\** folder.



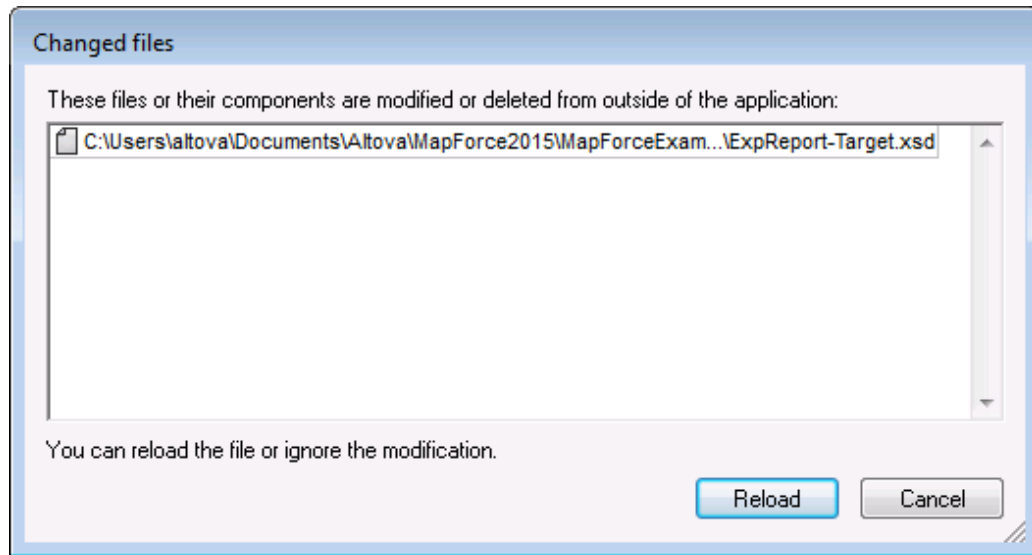


*Tut-ExpReport.mfd (MapForce Basic Edition)*

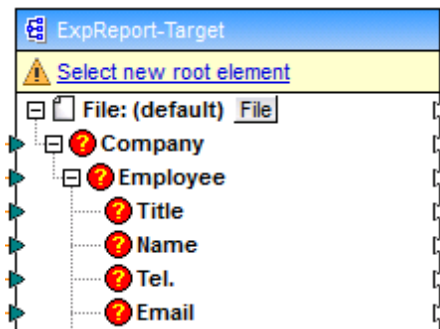
To understand how it works, do the following:

1. Open the **Tut-ExpReport.mfd** sample mapping.
2. Edit the **ExpReport-Target.xsd** schema outside MapForce so as to change the `Company` root element of the target schema to `Company-EU`. You do not need to close MapForce.
3. After you have changed the `Company` root element of the target schema to `Company-EU`, a "Changed files" prompt appears in MapForce.

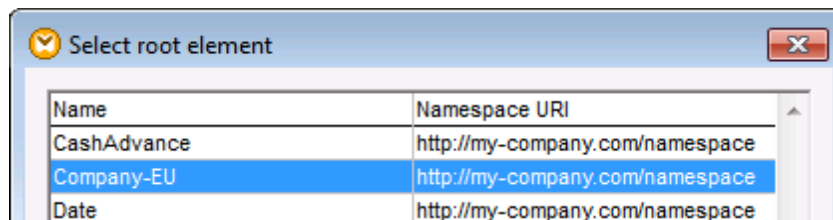




4. Click the **Reload** button to reload the updated Schema. Since the root element was deleted, the component displays multiple missing nodes.

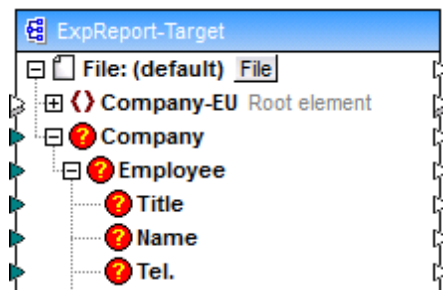


5. Click **Select new root element** at the top of the component. (You can also change the root element by right clicking the component header and selecting **Change Root Element** from the context menu.)

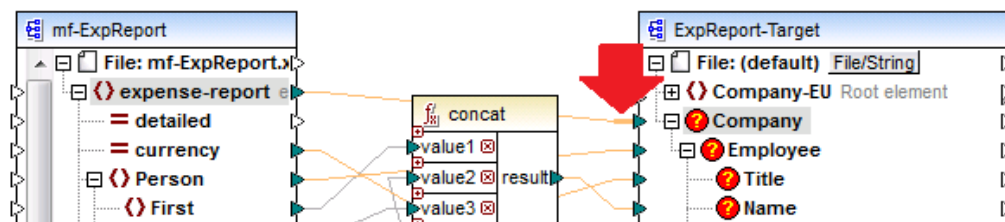


6. Select **Company-EU** as new root element and click **OK** to confirm. The **Company-EU** root element is now visible at the top of the component.

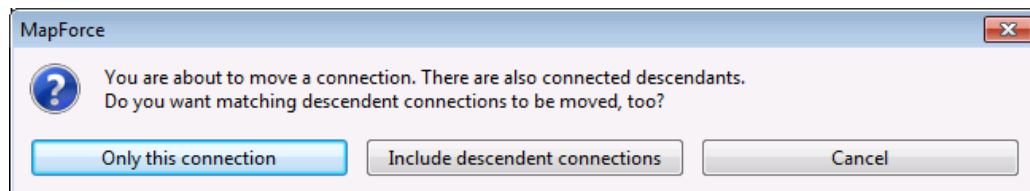




- Click the target stub of the connection that exists between the `expense-report` item of the source component and the `Company` item of the target component, and then drag-and-drop it on the `Company-EU` root element of the target component.



A notification dialog box appears.



- Click **Include descendent connections**. This instructs MapForce to re-map the correct child items under the new root element, and the mapping becomes valid again.

**Note:** If the node to which you are mapping has the same name as the source node but is in a different namespace, then the notification dialog box will contain an additional button: "Include descendants and map namespace". Clicking this button moves the child connections of the same namespace as the source parent node to the same child nodes under the different namespace node.

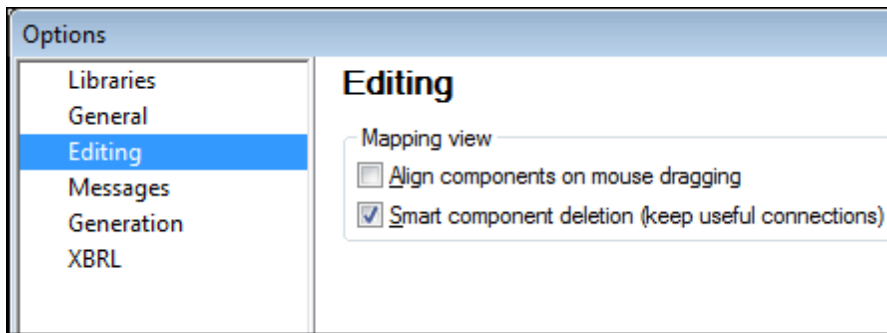
### 4.3.9 Keeping Connections After Deleting Components

You can decide what happens when you delete a component that has multiple (child) connections to another component, e.g. a filter or sort component. This is very useful if you want to keep all the child connections and not have to restore each one individually.

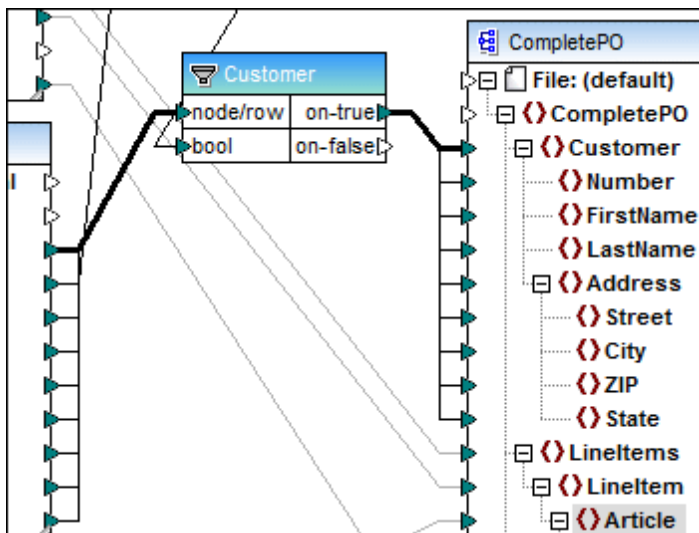
You can opt to keep/restore the child connections after the component is deleted, or to delete all child connections immediately.

Select **Tools | Options | Editing** (tab) to see the current setting. The default setting for the check box is **inactive**, i.e. "Smart component deletion (keep useful connections)" is disabled.

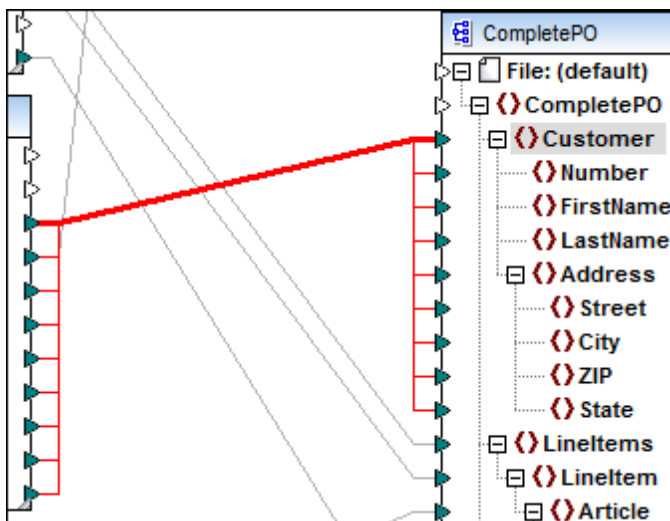




E.g. using the CompletePO.mfd mapping in the ...MapForceExamples folder, and the check box is active, the Customer filter is a [copy-all](#) connection with many connected child items, as shown below.



Deleting the Customer filter opens a prompt asking if you really want to delete it. If you select Yes, then the filter is deleted but all the child connectors remain.



Note that the remaining connectors are still selected (i.e. shown in red). If you want to delete them as well, hit the Del. key.



Clicking anywhere in the mapping area deselects the connectors.

If the "Smart component deletion..." check box is **inactive**, then deleting the filter will delete all child connectors immediately.

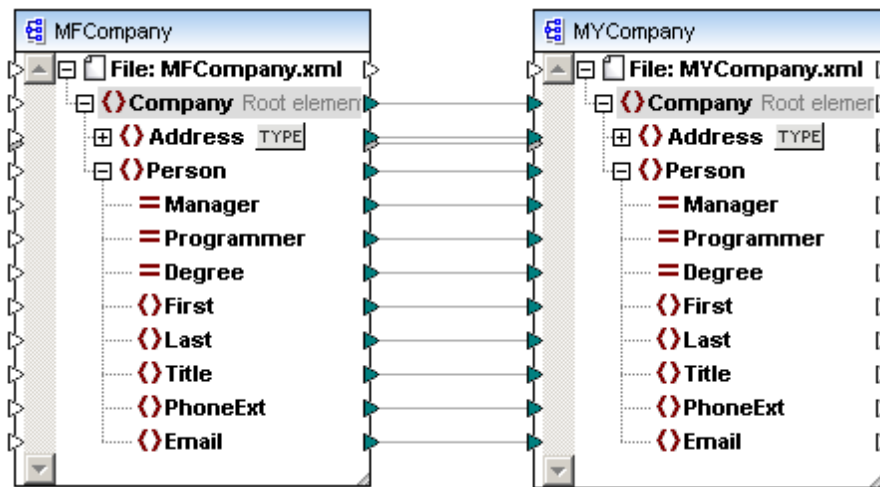
**Note:** If a filter component has both "on-true" and "on-false" outputs connected, then the connectors for both outputs will be retained.

#### 4.3.10 Dealing with Missing Items

Over time, it is likely that the structure of one of the components in a mapping may change e.g. elements or attributes are added/deleted to an XML schema. MapForce uses placeholder items to retain all the connectors, and any relevant connection data between components, when items have been deleted.

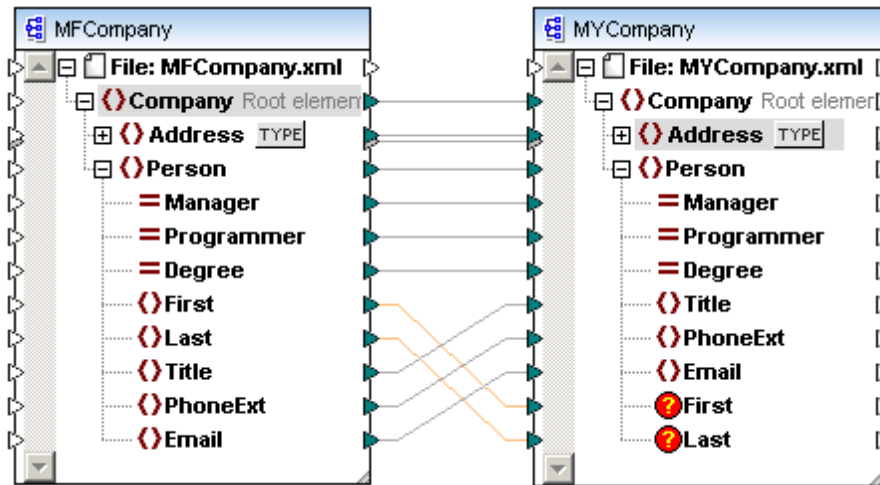
Example:

Using the **MFCompany.xsd** schema file as an example. The schema is renamed to **MyCompany.xsd** and a connector is created between the **Company** item in both schemas. This creates connectors for all child items between the components, if the Autoconnect Matching Children is active.



While editing **MyCompany.xsd**, in XMLSpy, the **First** and **Last** items in the schema are deleted. Returning to MapForce opens a Changed Files notification dialog box, prompting you to reload the schema. Clicking **Reload** updates the components in MapForce.

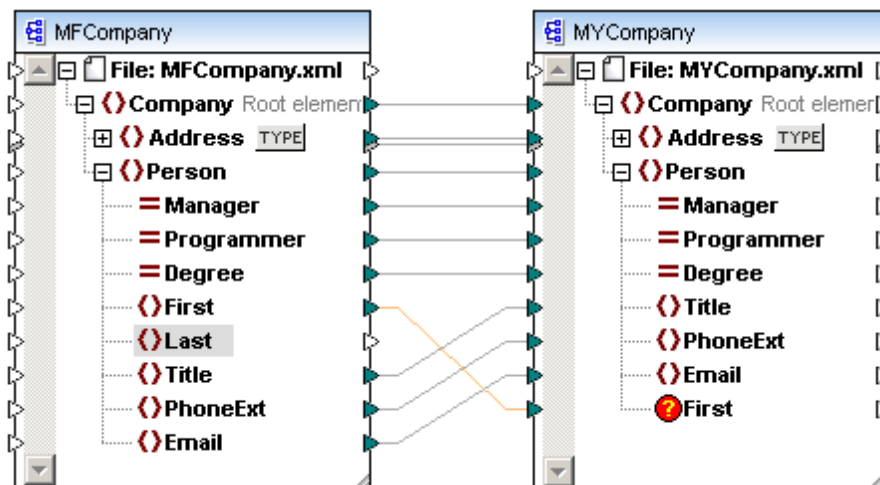




The deleted **items** and their **connectors** are now marked in the MyCompany component. You could now reconnect the connectors to other items if necessary, or delete the connectors.

Note that you can still preview the mapping (or generate code), but warnings will appear in the Messages window if you do so at this point. All connections to, and from, missing items are ignored during preview or code-generation.

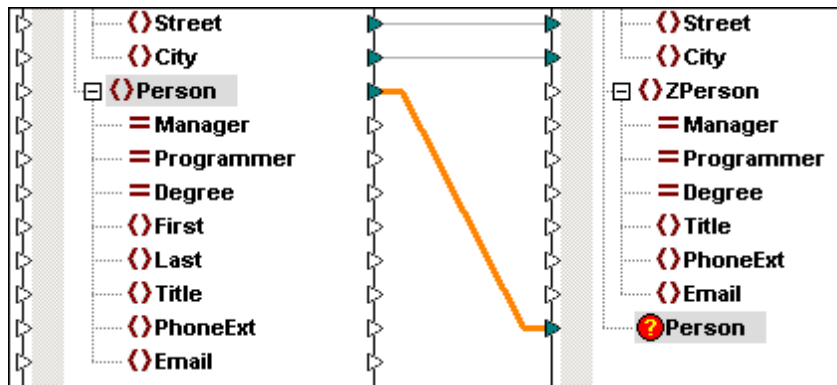
Clicking one of the highlighted connectors and deleting it, removes the "missing" item from the component, e.g. Last, in MyCompany.



### Renamed items

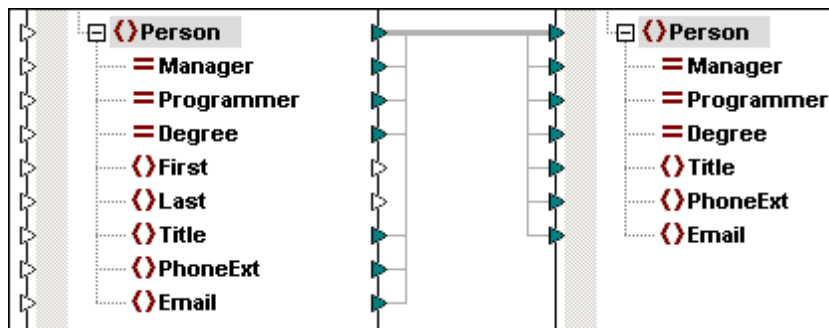
If a parent item is renamed e.g. Person to ZPerson, then the original parent item connector is retained and the child items and their connectors are deleted.





### "Copy all" connectors and missing items

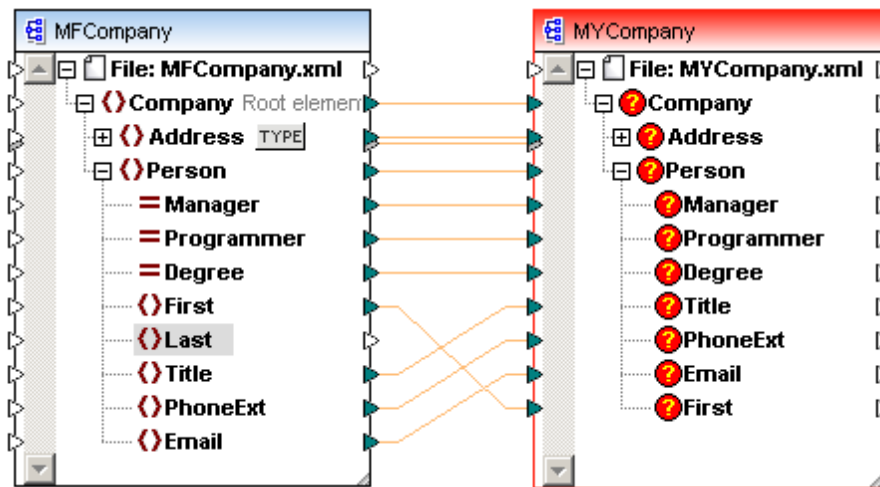
Copy all connections are treated in the same way as normal connections, with the only difference being that the connectors to the missing child items are not retained or displayed.



### Renamed or deleted component sources

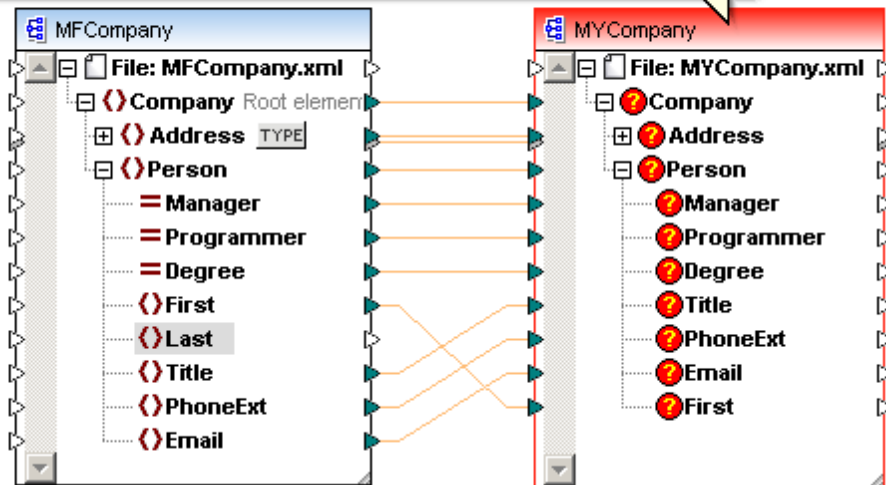
If the **data source** of a component i.e. schema has been renamed or deleted, then all items it contained are highlighted. The red frame around the component denotes that there is no valid connection to a schema and prevents preview and code generation.





Placing the mouse cursor over the highlighted component, opens a popup containing pertinent information.

This component does not have any valid structure information.  
Local file 'C:\2010\MapForceExamples\Tutorial\MYCompany.xsd' was not found.



Double-clicking the title bar of the highlighted component opens the Component Settings dialog box. Clicking the **Browse** button in the **Schema file** group allows you to select a different, or backed-up version of the schema. Please see "[Component](#)" in the Reference section for more information.

All valid/correct connections will be retained if you select a schema of the same structure.







# Chapter 5

---

## Designing Mappings



## 5 Designing Mappings

**Altova website:**  [Data integration tool](#)

This section describes how to design data mappings, and ways in which you can transform data on the mapping area. It also includes various considerations applicable to mapping design. Use the following roadmap for quick access to specific tasks or concepts:

I want to...	Read this topic...
Create or edit path references to miscellaneous schema, instance, and other files used by a mapping.	<a href="#">Using Relative and Absolute Paths</a>
Fine-tune the data mapping for specific needs (for example, influence the sequence of items in a target component).	<a href="#">Connection Types</a>
Use the output of a component as input of another component.	<a href="#">Chained mappings / pass-through components</a>
Process multiple files (for example, all files within a directory) in the same mapping, either as a source or a target.	<a href="#">Processing Multiple Input or Output Files Dynamically</a>
Pass an external value (such as a string parameter) to the mapping.	<a href="#">Supplying Parameters to the Mapping</a>
Get a string value out of the mapping, instead of a file.	<a href="#">Returning String Values from a Mapping</a>
Store some mapping data temporarily for later processing (similar to variables in a programming language).	<a href="#">Using Variables</a>
Sort data in ascending or descending order.	<a href="#">Sorting Data</a>
Filter nodes/rows based on specific criteria, or process values conditionally.	<a href="#">Filters and Conditions</a>
Merge data from multiple sources with different schema.	<a href="#">Merging Data from Multiple Schemas</a>
Process key-value pairs, for example, to convert months from numerical representation (01, 02, and so on) to text representation (January, February, and so on).	<a href="#">Using Value-Maps</a>
Learn how to avoid undesired results when designing complex mappings.	<a href="#">Mapping rules and strategies</a>

Importantly, MapForce additionally includes an extensive built-in function library (see [Function](#)



[Library Reference](#)) to help you with a wide array of processing tasks. When the built-in library is not sufficient, you can always build your own custom functions in MapForce, or re-use external XSLT files. For further information, see [Using Functions](#).



## 5.1 Using Relative and Absolute Paths

A mapping design file (\*.mfd) may have references to several schema and instance files. The schema files are used by MapForce to determine the structure of the data to be mapped, and to validate it. The instance files, on the other hand, are required to read, preview, and validate the source data against the schema.

All references to files used by a mapping design are created by MapForce when you add a component to the mapping. However, you can always set or change such path references manually if required.

This section provides instructions for setting or changing the path to miscellaneous file types referenced by a mapping, and the implications of using relative versus absolute paths.

### 5.1.1 Using Relative Paths on a Component

The Component Settings dialog box (illustrated below for an XML component) provides the option to specify either absolute or relative paths for various files which may be referenced by the component:

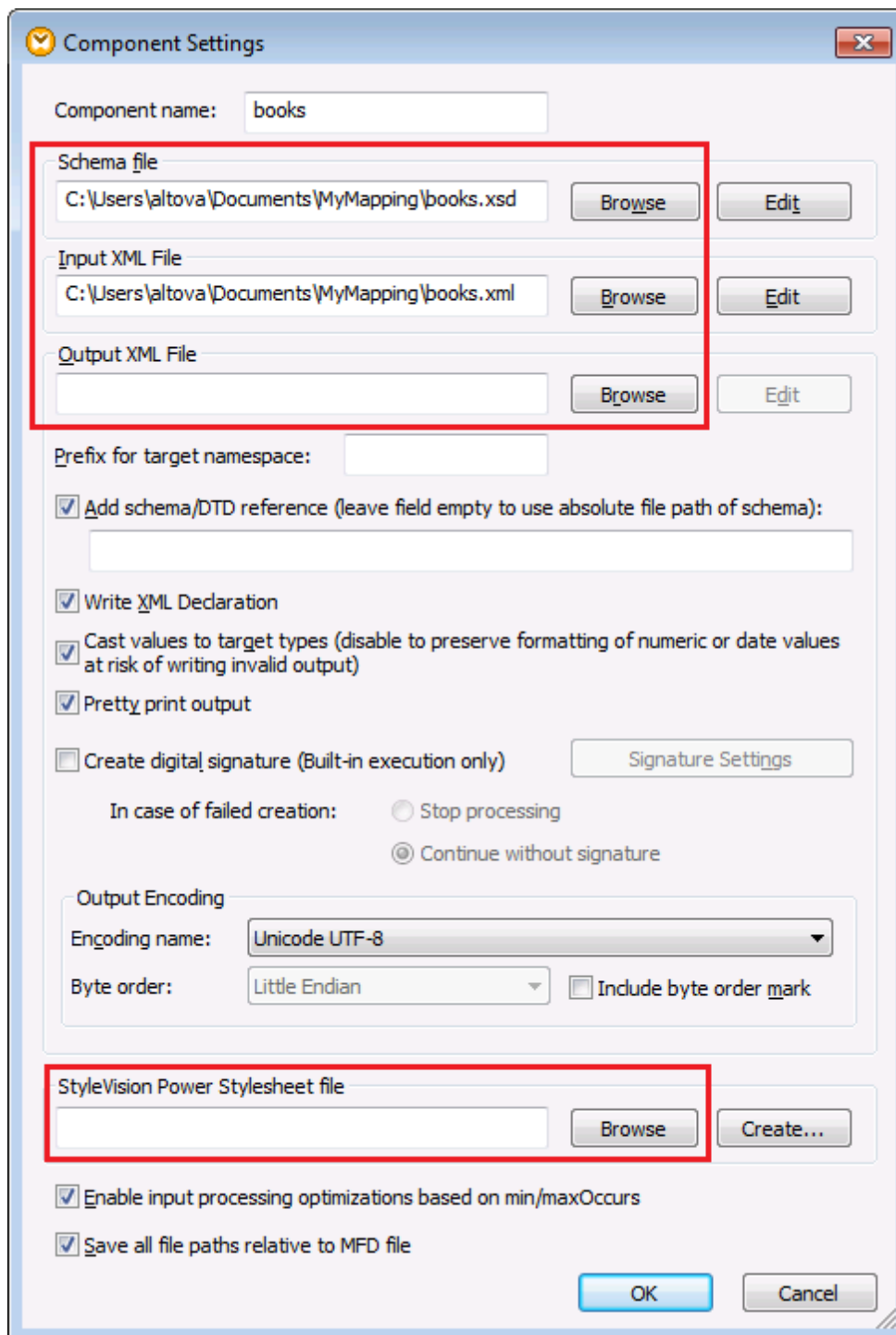
- Input files (that is, files from which MapForce reads data)
- Output files (that is, files to which MapForce writes data)
- Schema files (applicable to components which have a schema)
- Structure files (applicable to components which may have a complex structure, such as input or output parameters of user-defined functions, or variables)
- StyleVision Power Stylesheet (\*.sps) files, used to format data for outputs such as PDF, HTML and Word.

You can enter relative paths directly in the relevant text boxes (shown enclosed in a red frame in the image below).

Before entering relative file paths, make sure to save the mapping file (.mfd) first. Otherwise, all relative paths are resolved against the personal application folder of Windows (Documents \Altova\MapForce2018), which may not be the intended behavior.

You can also instruct MapForce to save all above-mentioned file paths relative to the mapping .mfd file. In the sample image below, notice the option **Save all file paths relative to MFD file**. If the check box is enabled (which is the default and recommended option), the paths of any files referenced by the component will be saved relative to the path of the mapping design file (.mfd). This affects all files referenced by the component (shown enclosed in a red frame in the image).





Component Settings dialog box

Although the component illustrated above is an XML component, the setting **Save all file paths relative to MFD file** works in the same way for the following files:

- Structure files used by complex input or output parameters of user-defined functions and



- variables of complex type
- Input or output flat files \*
- Schema files referenced by database components which support XML fields \*
- Input or output XBRL, FlexText, EDI, Excel 2007+, JSON files \*\*

\* MapForce Professional and Enterprise Edition

\*\* MapForce Enterprise Edition only

Taking the component above as an example, if the .mfd file is in the same folder as the **books.xsd** and **books.xml** files, the paths will be changed as follows:

**C:\Users\altova\Documents\MyMapping\books.xsd** will change to **books.xsd**

**C:\Users\altova\Documents\MyMapping\books.xml** will change to **books.xml**

Paths that reference a non-local drive or use a URL will not be made relative.

When the check box is enabled, MapForce will also keep track of the files referenced by the component if you save the mapping to a new folder using the **Save as** menu command. Also, if all files are in the same directory as the mapping, path references will not be broken when you move the entire directory to a new location on the disk.

Using relative paths (and, therefore, enabling the **Save all file paths relative to MFD file** check box) may be important in many cases, for example:

- The location of the mapping on your operating system is likely to change in future.
- The mapping is stored in a directory which is under source control (using a version control system such as TortoiseSVN, for example).
- You intend to deploy the mapping for execution to a different machine or even to a different operating system.

If the **Save all file paths relative to MFD file** check box is disabled, saving the mapping does not modify the file paths (that is, they remain as they appear in the Component Settings dialog box).

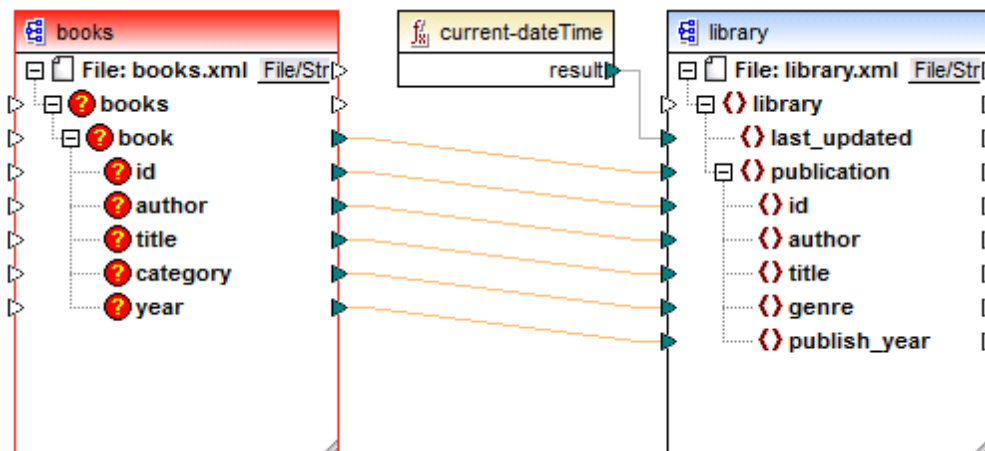
## 5.1.2 Fixing Broken Path References

When you add or change a file reference in a mapping, and the path cannot be resolved, MapForce displays a warning message. This way, MapForce diminishes the chance for broken path references to happen. Nevertheless, broken path references may still occur in cases such as:

- You use relative paths, and then move the mapping file to a new directory without moving the schema and instance files.
- You use absolute paths to files in the same directory as the mapping file, and then move the directory to another location.

When this happens, MapForce highlights the component in red, for example:





Broken path reference

The solution in this case is to double-click the component header and update any broken path references in the **Component Settings** dialog box (see also [Changing the Component Settings](#) ).

### 5.1.3 Paths in Various Execution Environments

If you generate code from mappings, the generated files are no longer run by MapForce. Instead, the mappings are run by the target environment you have chosen (for example, RaptorXML Server). The implication is that, for the mapping to run successfully, any relative paths must be meaningful in the environment where the mapping runs.

Consequently, when the mapping uses relative paths to instance or schema files, consider the base path to be as follows for each target language:

Target language	Base path
XSLT/XSLT2	Path of the XSLT file.
XQuery*	Path of the XQuery file.
C++, C#, Java*	Working directory of the generated application.
BUILT-IN* (when previewing the mapping in MapForce)	Path of the mapping (.mfd) file.
BUILT-IN* (when running the mapping with MapForce Server)	The current working directory.
BUILT-IN* (when running the mapping with MapForce Server under FlowForce Server control)	The working directory of the job or the working directory of FlowForce Server.

\* Languages available in MapForce Professional and Enterprise editions



If required, you can instruct MapForce to convert all paths from relative to absolute when generating code for a mapping. This option might be useful if you run the mapping code on the same operating system, or perhaps on another operating system where any absolute path references used by the mapping can still be resolved.

To convert all paths to absolute in the generated code, select the **Make paths absolute in generated code** check box, on the Mapping Settings dialog box (see [Changing the Mapping Settings](#) ).

When you generate code and the check box is selected, MapForce resolves any relative paths based on the directory of the mapping file (.mfd), and makes them absolute in the generated code. This setting affects the path of the following files:

- Input and output instance files for all file-based component kinds

When the check box is not selected, the file paths will be preserved as they are defined in the component settings.

#### 5.1.4 Copy-Paste and Relative Paths

When you copy a component from a mapping and paste it into another, a check is performed to ensure that relative paths of schema files can be resolved against the folder of the destination mapping. If the path cannot be resolved, you will be prompted to make the relative paths absolute by means of the folder of the source mapping. It is recommended to save the destination mapping first, otherwise relative paths are resolved against the personal application folder.



## 5.2 Connection Types

When you create a mapping connection (and both the source and the target item have child items), you can optionally choose the type of the connection to be one of the following.

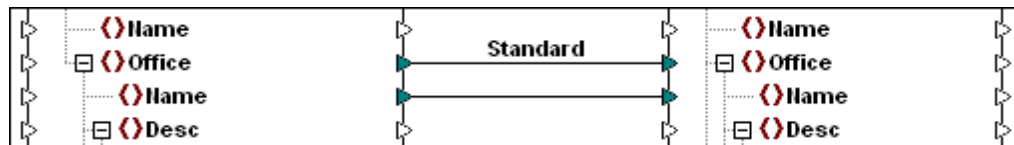
- Target Driven (Standard)
- Source Driven (Mixed Content)
- Copy-All (Copy Child Items).

The connection type determines the sequence of children items in the output generated by the mapping. This section provides information about each connection type and the scenarios when they are useful.

### 5.2.1 Target-driven connections

When a connection is "target-driven" (or "standard"), the sequence of child nodes in the mapping output is determined by the sequence of nodes in the target schema. This connection type is suitable for most mapping scenarios and is the default connection type used in MapForce.

On a mapping, target-driven connections are shown with a solid line.



Target-driven connections might not be suitable when you want to map XML nodes that contain mixed context (character data as well as child elements), for example:

```
<p>This is our <i>best-selling</i> product.</p>
```

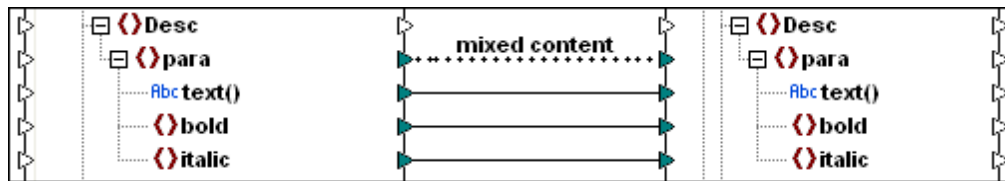
With mixed content, it is likely that you want to preserve the sequence of items as they appear in the source file, in which case a source-driven connection is recommended (see [Source-driven connections](#)).

### 5.2.2 Source-driven connections

Source-driven (Mixed Content) mapping enables you to automatically map text and child nodes in the same sequence that they appear in the XML **source** file.

- Mixed content text node content is supported/mapped.
- The sequence of child nodes is dependent on the source XML instance file.





Mixed content mappings are shown with a dotted line.

Source-driven / mixed content mapping can also be applied to XML schema **complexType** items. Child nodes will then be mapped according to their sequence in the XML source file.

Source-driven / mixed content mapping supports:

Mappings from

- As **source** components:
  - XML schema complexTypes (including mixed content, i.e. mixed=true)
- As **target** components:
  - XML schema complexTypes (including mixed content), Note: CDATA sections are treated as text.

### 5.2.2.1 Mapping mixed content

The files used in the following example (**Tut-OrgChart.mfd**, **Tut-OrgChart.mfd.xml**, **Tut-OrgChart.mfd.xsd**, **Tut-Person.xsd**) are available in the [...\MapForceExamples\Tutorial\](#) folder.

#### Source XML instance

A portion of the **Tut-OrgChart.xml** file used in this section is shown below. Our area of concern is the mixed content element "para", along with its child nodes "bold" and "italic".

The `para` element also contains a Processing Instruction (`<?sort alpha-ascending?>`) as well as Comment text (`<!--Company details... -->`) which can also be mapped, as shown below.



```

<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2005 sp2 U (http://www.altova.com) by Mr. Nobody (Altova GmbH) -->
<OrgChart xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Tut-OrgChart.xsd">
  <CompanyLogo href="nanonull.gif"/>
  <Name>Organization Chart</Name>
  <Office>
    <Name>Nanonull, Inc.</Name>
    <Desc>
      <para>The company was established in<b>Vereno</b>in 1995. Nanonull
develops nanoelectronic technologies for<i>multi-core processors.</i>February 1999
saw the unveiling of the first prototype <b>Nano-grid.</b>The company hopes to expand
its operations <i>offshore</i>to drive down operational costs.
      <?sort alpha-ascending?>
      <!--Company details: location and general company information.-->
    </para>
    <para>White papers and further information will be made available in the near future.
    </Desc>
  </Office>
</OrgChart>

```

Note the sequence of the text and bold/italic nodes in the XML instance file:

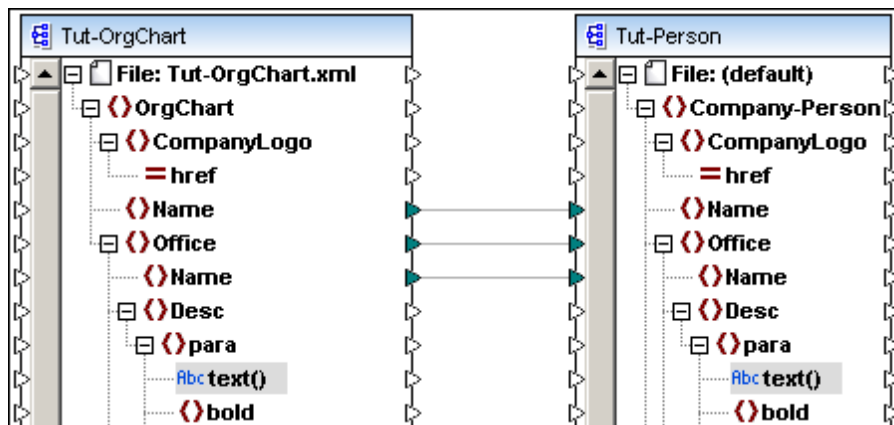
```

<para> The company...
  <b>Vereno</b>in 1995 ...
  <i>multi-core...</i>February 1999
  <b>Nano-grid.</b>The company ...
  <i>offshore...</i>to drive...
</para>

```

### Initial mapping

The initial state of the mapping when you open **Tut-Orgchart.mfd** is shown below.



### Output of above mapping

The result of the initial mapping is shown below: Organization Chart as well as the individual office names have been output.



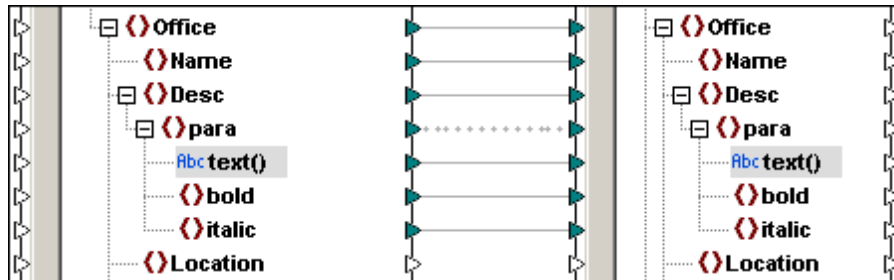
```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <Company-Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNames
3    <Name>Organization Chart</Name>
4    <Office>
5      <Name>Nanonull, Inc.</Name>
6    </Office>
7    <Office>
8      <Name>Nanonull Europe, AG</Name>
9    </Office>
10 </Company-Person>
11

```

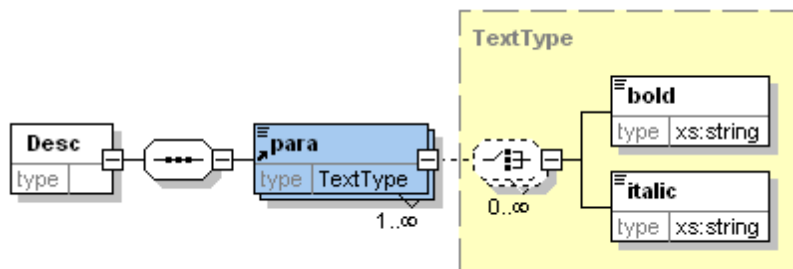
### Mapping the para element

The image below shows an example of mixed content mapping. The para element is of mixed content, and the connector is shown as a **dotted** line to highlight this. The **text()** node contains the textual data and needs to be mapped for the text to appear in the target component.



To annotate (add a label to) any connection, right-click it and select **Properties** (see [Annotating Connections](#) ).

The image below shows the content model of the Description element (Desc) of the **Tut-OrgChart.xsd** schema file. This definition is identical in both the source and target schemas used in this example.



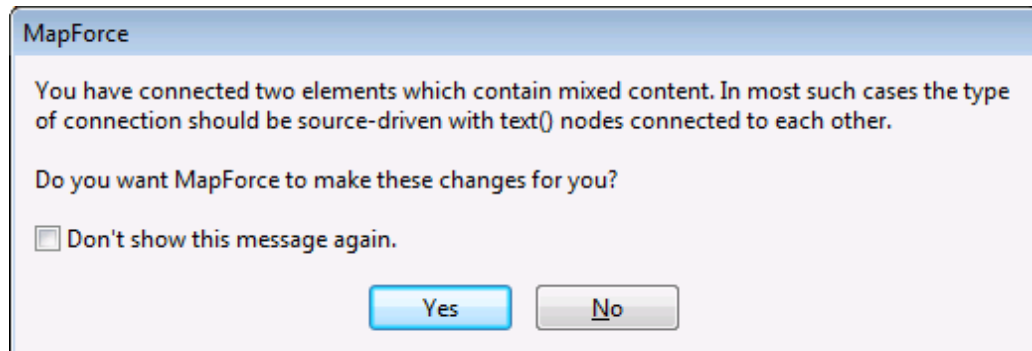
Note the following properties of the **para** element in the Content model:

- **para** is a complexType with mixed="true", of type "TextType"
- **bold** and **italic** elements are both of type "xs:string", they have not been defined as recursive in this example, i.e. neither **bold**, nor **italic** are of type "TextType"
- **bold** and **italic** elements can appear any number of times in any sequence within **para**
- any number of text nodes can appear within the **para** element, interspersed by any number of **bold** and **italic** elements.



**To create mixed content connections between items:**

1. Select the menu option **Connection | Auto Connect Matching Children** to activate this option, if it is not currently activated.
2. Connect the **para** item in the source schema, with the **para** item in the target schema. A message appears, asking if you would like MapForce to define the connectors as source driven.

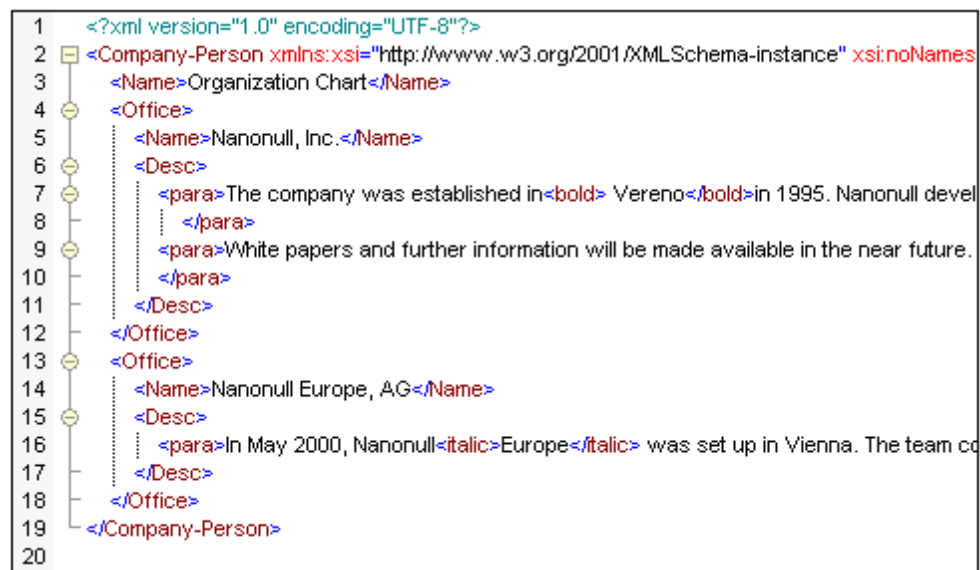



3. Click Yes to create a mixed content connection.

**Note:** Para is of mixed content, and makes the message appear at this point. The mixed-content message also appears if you only map the para items directly, without having the autoconnect option activated.

All child items of para have been connected. The connector joining the para items is displayed as a dotted line, to show that it is of type mixed content.

4. Click the Output tab to see the result of the mapping.



5. Click the word **Wrap** icon  in the Output tab icon bar, to view the complete text in the Output window.



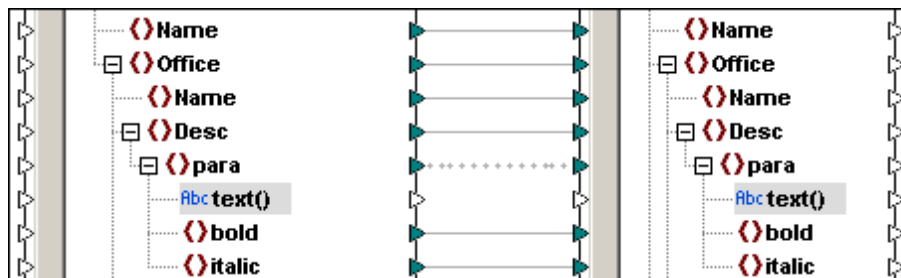


The mixed content text of each office description has been mapped correctly; the text, as well as the bold and italic tag content, have been mapped as they appear in the XML **source** file.

- Switch back to the Mapping view.

#### To remove text nodes from mixed content items:

- Click the **text()** node connector and press Del. to delete it.



- Click the Output tab to see the result of the mapping.





Result:

- all **text** nodes of the para element have been removed.
- mapped bold and italic text content remain
- the bold and italic item **sequence** still follows that of the source XML file.

To map the Processing Instructions and Comments:

1. Right-click the mixed content connection, and select **Properties**.
2. Under **Source-Drive (Mixed content)**, select the **Map Processing Instructions** and **Map Comments** check boxes.

### 5.2.2.2 Mixed content example

The following example is available as "ShortApplicationInfo.mfd" in the [...MapForceExamples](#) folder.

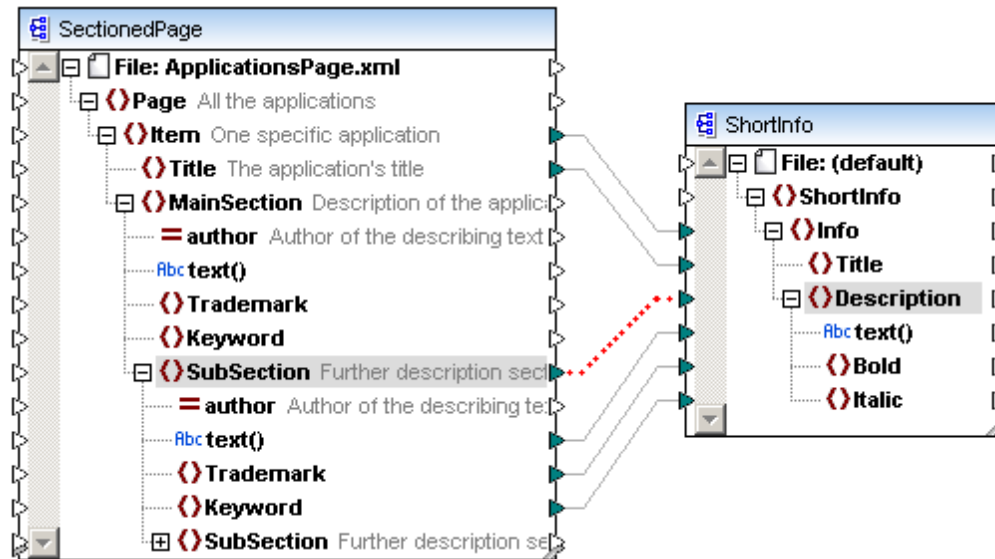
A snippet of the XML source file for this example is shown below.

```
<Page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="SectionedPage.xsd">
  <Item>
    <Title>XMLSpy</Title>
    <MainSection author="altova">
      Altova <Trademark>XMLSpy</Trademark>
      <SubSection>Altova <Trademark>XMLSpy</Trademark> 2005 Enter
is the industry standard <Keyword>XML</Keyword> development environment
editing, debugging and transforming all <Keyword>XML</Keyword> technolo
automatically generating runtime code in multiple programming languages
    </MainSection>
  </Item>
```

The mapping is shown below. Please note the following:



- The "SubSection" item connector is of mixed content, and is mapped to the Description item in the target XML/schema.
- The text() nodes are mapped to each other
- Trademark text is mapped to the Bold item in the target
- Keyword text is mapped to the Italic item in the target



### Mapping result

The mixed content text of each description has been mapped correctly; the text, as well as the bold and italic tag content, have been mapped as they appear in the XML source file.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <ShortInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="
   C:/PROGRA~1/Altova/MapForce2005/MapForceExamples/ShortInfo.xsd">
3    <Info>
4      <Title>XMLSpy</Title>
5      <Description>Altova <Bold>XMLSpy</Bold> 2005 Enterprise Edition is the industry standard
   <Italic>XML</Italic> development environment for modeling, editing, debugging and transforming
   all <Italic>XML</Italic> technologies, then automatically generating runtime code in multiple
   programming languages.</Description>
6    </Info>

```

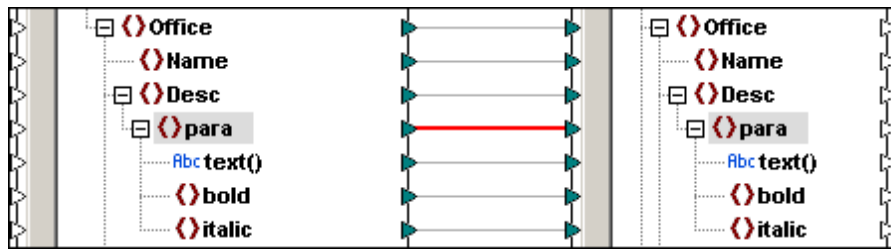
### 5.2.2.3 Using standard connections on mixed content items

As mentioned before, source-driven (not standard) connections are normally used when mapping data from mixed content nodes. Otherwise, the resulting output may be undesirable. To see the consequences of using a standard (target-driven) connection when mapping data from a mixed content node, follow the steps below:

1. Open the mapping **Tut-OrgChart.mfd** from the [<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\](#) folder.
2. Create a connection between the `para` node in the source and the `para` node in the target. A message appears, asking if you would like MapForce to define the connections

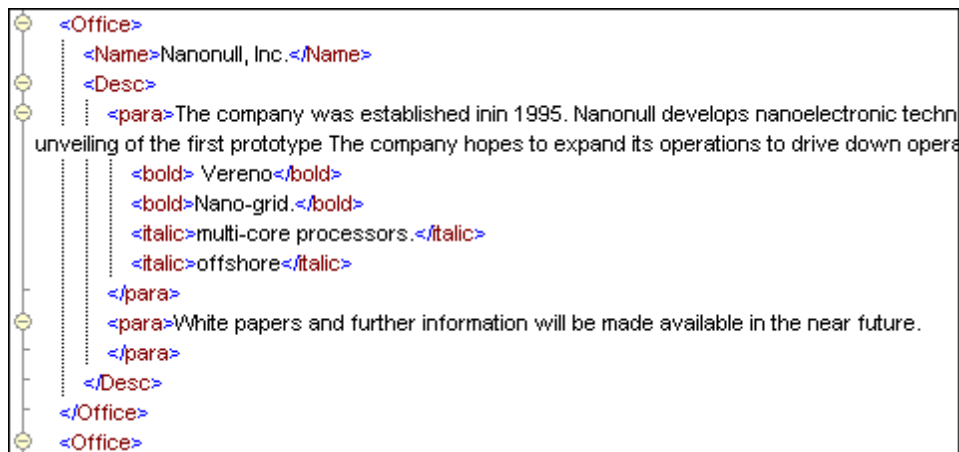


as source-driven. Click **No** (this disregards the MapForce suggestion and creates a standard connection).



**Note:** Make sure that the connection is standard (target-driven), as shown above. If a [Copy-All](#) connection is created automatically, right-click the connection, and select **Target Driven (Standard)** from the context menu.

3. Click the **Output** tab to see the result of the mapping.



As illustrated above, mapping mixed content nodes using standard connections produces the following result:

- The content of the `text()` source item is copied to the target; however, the sequence of child nodes (`bold` and `italic`, in this case) in the output corresponds to the sequence in the target XML schema. In other words, the child nodes (`bold` and `italic`, in this case) appear after the mixed content node `text`.
- For each `para` element, MapForce has mapped the `text()` node first, then all `bold` items, and, finally, all `italic` items. As a result, multiple `bold` and `italic` items appear stacked on each other. Note that the content of each item is mapped if a connection exists to it from the source.

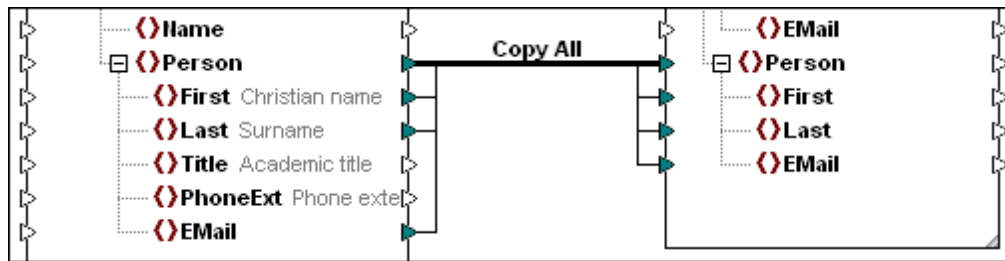
### 5.2.3 Copy-All Connections

Copy-All connections map data between complex structures (nodes with children items) that are very similar or identical. The main benefit of "Copy-All" connections is that they simplify the mapping workspace (one "thick" connection is created instead of multiple).

On the mapping, a "Copy-All" connection appears as a single bold line (with input and output



"forks" for each child item) that connects two identical or similar structures.



*Copy-All connection*

When you draw a mapping connection between two structures on the mapping, MapForce creates a "Copy-All" connection automatically if it detects that the source and target structure are assignment compatible (that is, when both structures are either of the same type, or the target is a subtype of the source type). At mapping runtime, all instance data will be copied from the source to the target recursively, including children.

To create a "Copy-All" connection manually, right-click an existing connection between two similar nodes with child items, and select **Copy-All (Copy Child Items)** from the context menu.

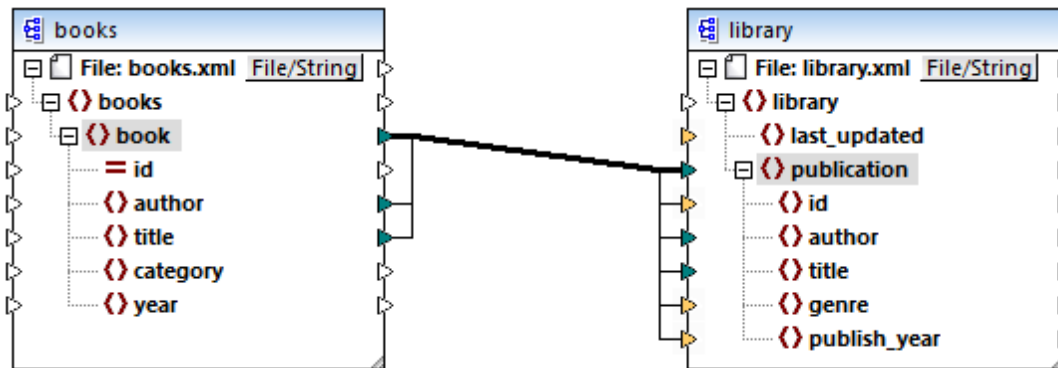
Note the following:

- In contexts where a "Copy-All" connection is not meaningful or not supported, it is not possible to create this kind of connection manually.
- A "Copy-All" connection cannot be created to the `root` element of an XML/Schema component.
- When creating "Copy-All" connections between a schema and a parameter of a user-defined function, the two components must be based on the same schema. It is not necessary that they both have the same root elements, however.

For an example of a "Copy-All" connection created manually, take the following steps:

1. Create a new mapping.
2. On the **Insert** menu, click **XML Schema/File** and browse for the **books.xml** file located in the folder `<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\`.
3. On the **Insert** menu, click **XML Schema/File** and browse for the **library.xsd** file located in the folder `<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\`.
4. Draw a mapping connection between the `book` node of the "books" component to the `publication` node of the "library" component.
5. Right-click the new connection, and select **Copy-All (Copy Child Items)** from the context menu.

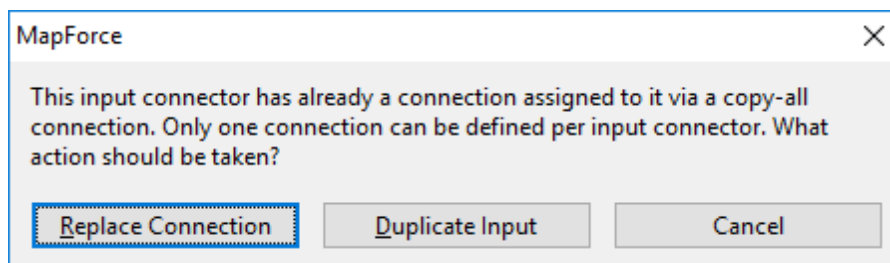




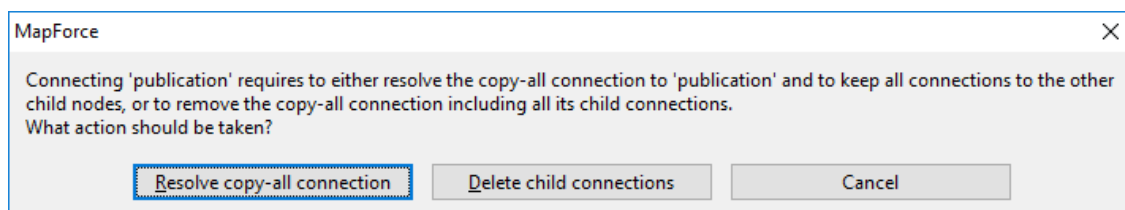
If there are slight differences between the source and the target structures, the "Copy-All" connection will enumerate, at mapping runtime, the source items (such as elements and attributes) and will copy only those that exist in the target type. This is repeated recursively.

For example, in the mapping above, only two child items are identical between the two structures (author and title) and thus they are mapped to the target. The item `id` is not included automatically because it is an attribute in the source and an element in the target. If you need to map, for example, `category` to `genre`, the "Copy-All" connection is no longer possible, because these are different items.

When an input connector (the small triangle to the side of the component) receives a "Copy-All" connection, it cannot accept any other connections. In the example above, if you attempt to create a connection between `category` and `genre`, MapForce prompts you to either replace it, or duplicate the input.



Duplicating input is meaningful only if you want the target to accept data from more than one input, which is not required here (see also [Duplicating Input](#)). If you choose to replace the "Copy-All" connection, a message box prompts you again to either resolve or delete the "Copy-All" connection.



Click **Resolve copy-all connection** if you want to replace the "Copy-All" connection by standard



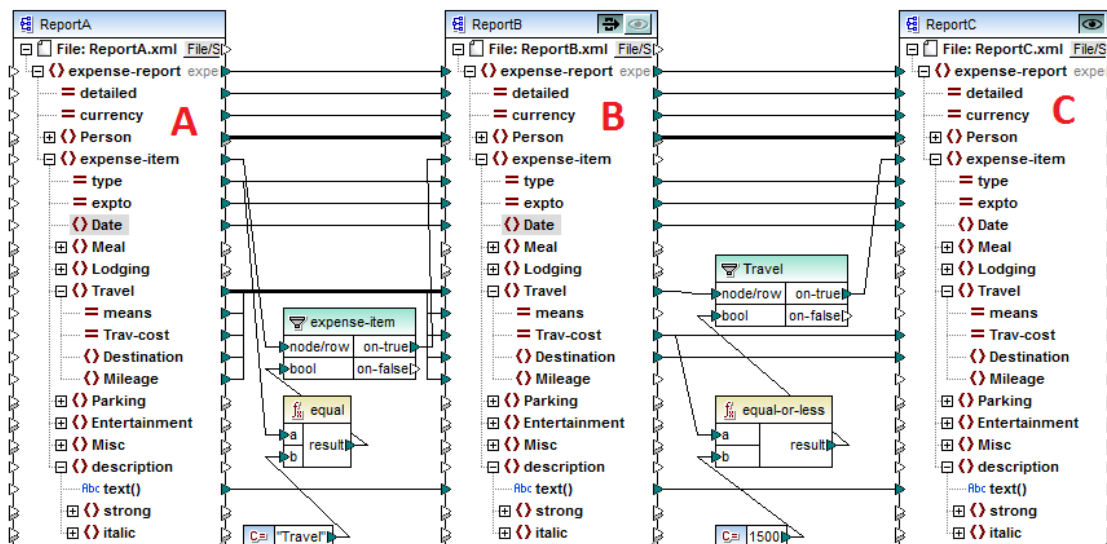
individual target-driven connections to corresponding child items. If you prefer to remove the "Copy-All" connection completely, click **Delete child connections**.



## 5.3 Chained Mappings

MapForce supports mappings that consist of multiple components in a mapping chain. Chained mappings are mappings where at least one component acts both as a source and a target. Such a component creates output which is later used as input for a following mapping step in the chain. Such a component is called an "intermediate" component.

For example, the mapping illustrated below shows an expense report (in XML format) that is being processed in two stages. The part of the mapping from A to B filters out only those expenses that are marked as "Travel". The mapping from B to C filters out only those "Travel" expenses that have a travel cost less than 1500. Component B is the "intermediate" component, as it has both input and output connections. This mapping is available at the following path: **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\ChainedReports.mfd**.



*ChainedReports.mfd*

Chained mappings introduce a feature called "pass-through". "Pass-through" is a preview capability allowing you to view the output produced at each stage of a chained mapping in the Output window. For example, in the mapping above, you can preview and save the XML output resulting from A to B, as well as the XML output resulting from B to C.

**Note:** The "pass-through" feature is available only for file-based components (for example, XML, CSV, and text). Database components can be intermediate, but the pass-through button is not shown. The intermediate component is always regenerated from scratch when previewing or generating code. This would not be feasible with a database as it would have to be deleted prior to each regeneration.

If the mapping is executed by MapForce Server, or by generated code, then the full mapping chain is executed. The mapping generates the necessary output files at each step in the chain, and the output of a step of a mapping chain is forwarded as input to the following mapping step.

It is also possible for intermediate components to generate dynamic file names. That is, they can accept connections to the "File:" item from the mapping, provided that the component is



configured correspondingly. For more information, see [Processing Multiple Input or Output Files Dynamically](#).


### **Preview button**

Both the component B and the component C have preview buttons. This allows you to preview in MapForce the intermediate mapping result of B, as well as the final result of the chained mapping. Click the preview button of the respective component, then click Output to see the mapping result.


"Intermediate" components with the pass-through button active cannot be previewed. Their preview button is automatically disabled, because it is not meaningful to preview and let data pass through at the same time. To see the output of such a component, first click the "pass-through" button to deactivate it, and then click the preview button.

### **Pass-through button**

The intermediate component B has an extra button in the component title bar called "pass-through".

If the pass-through button is **active** , MapForce maps all data into the preview window in one go; from component A to component B, then on to component C. Two result files will be created:

- the result of mapping component A to intermediate component B
- the result of the mapping from the intermediate component B, to target component C.

If the pass-through button is **inactive** , MapForce will execute only parts of the full mapping chain. Data is generated depending on which preview buttons are active:

- If the preview button of component B is active, then the result of mapping component A to component B is generated. The mapping chain actually stops at component B. Component C is not involved in the preview at all.
- If the preview button of component C is active, then the result of mapping intermediate component B to the component C is generated. Because pass-through is inactive, automatic chaining has been interrupted for component B. Only the right part of the mapping chain is executed. Component A is not used.

When the "pass-through" button is inactive, it is important that the intermediate component has identical file names in the "Input XML File" and "Output XML File" fields. This ensures that the file generated as output when you preview the portion of the mapping between A and B is used as input when you preview the portion of the mapping between B and C. Also, in generated code, or in MapForce Server execution, this ensures that the mapping chain is not broken.

As previously mentioned, if the mapping is executed by MapForce Server, or by generated code, then the output of all components is generated. In this case, the settings of the pass-through button of component B, as well as the currently selected preview component, are disregarded. Taking the mapping above as example, two result files will be generated, as follows:

1. The output file resulting from mapping component A to B
2. The output file resulting from mapping component B to C.



The following sections, [Example: Pass-Through Active](#) and [Example: Pass-Through Inactive](#), illustrate in more detail how the source data is transferred differently when the pass-through button is active or inactive.

### 5.3.1 Example: Pass-Through Active

The mapping used in this example (**ChainedReports.mfd**) is available in the **<Documents> \Altova\MapForce2018\MapForceExamples\Tutorial\** folder. This mapping processes an XML file called **ReportA.xml** that contains travel expenses and looks as shown below. For simplicity, the namespace declaration and some `expense-item` elements have been omitted:


```
<?xml version="1.0" encoding="UTF-8"?>
<expense-report currency="USD" detailed="true">
  <Person>
    <First>Fred</First>
    <Last>Landis</Last>
    <Title>Project Manager</Title>
    <Phone>123-456-78</Phone>
    <Email>f.landis@nanonull.com</Email>
  </Person>
  <expense-item type="Travel" expto="Development">
    <Date>2003-01-02</Date>
    <Travel Trav-cost="337.88">
      <Destination/>
    </Travel>
    <description>Biz jet</description>
  </expense-item>
  <expense-item type="Lodging" expto="Sales">
    <Date>2003-01-01</Date>
    <Lodging Lodge-cost="121.2">
      <Location/>
    </Lodging>
    <description>Motel mania</description>
  </expense-item>
  <expense-item type="Travel" expto="Marketing">
    <Date>2003-02-02</Date>
    <Travel Trav-cost="2000">
      <Destination/>
    </Travel>
    <description>Hong Kong</description>
  </expense-item>
</expense-report>
```

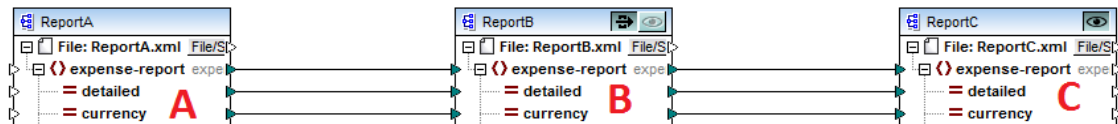
*ReportA.xml*

The goal of the mapping it to produce, based on the file above, two further reports:

- **ReportB.xml** - this report should contain only those travel expenses that are of type "Travel".
- **ReportC.xml** - this report should contain only those travel expenses that are of type "Travel" and do not exceed 1500.



To achieve this goal, the intermediate component of the mapping (component B) has the pass-through button  active, as shown below. This causes the mapping to be executed in stages: from A to B, and then from B to C. The output created by the intermediate component will be used as input for the mapping between B and C.



The names of generated output files at each stage in the mapping chain is determined by the settings of each component. (To open the component settings, right-click it, and then select **Properties** from the context menu). Namely, the first component is configured to read data from an XML file called **ReportA.xml**. Because this is a source component, the **Output XML File** field is irrelevant and it was left empty.

*Settings of the source component*

As shown below, the second component (**ReportB**) is configured to create an output file called **ReportB.xml**. Notice that the **Input XML File** field is grayed out. When pass-through is active (as in this example), the **Input XML File** field of the intermediate component is automatically deactivated. An input file name need not exist for the mapping to execute, because the output created at this stage in the mapping is stored in a temporary file and reused further in the mapping. Also, if an **Output XML File** is defined (as illustrated below), then it is used for the file name of the intermediate output file. If no **Output XML File** is defined, a default file name will be automatically used.



The screenshot shows a configuration window for a component named "ReportB". It has four sections: "Component name" with a text box containing "ReportB"; "Schema file" with a text box containing "ExpenseReport.xsd" and "Browse" and "Edit" buttons; "Input XML File" with a text box containing "ReportB.xml" and "Browse" and "Edit" buttons; and "Output XML File" with a text box containing "ReportB.xml" and "Browse" and "Edit" buttons.

*Settings of the intermediate component*

Finally, the third component is configured to produce an output file called **ReportC.xml**. The **Input XML File** field is irrelevant here, because this is a target component.

The screenshot shows a configuration window for a component named "ReportC". It has four sections: "Component name" with a text box containing "ReportC"; "Schema file" with a text box containing "ExpenseReport.xsd" and "Browse" and "Edit" buttons; "Input XML File" with an empty text box and "Browse" and "Edit" buttons; and "Output XML File" with a text box containing "ReportC.xml" and "Browse" and "Edit" buttons.

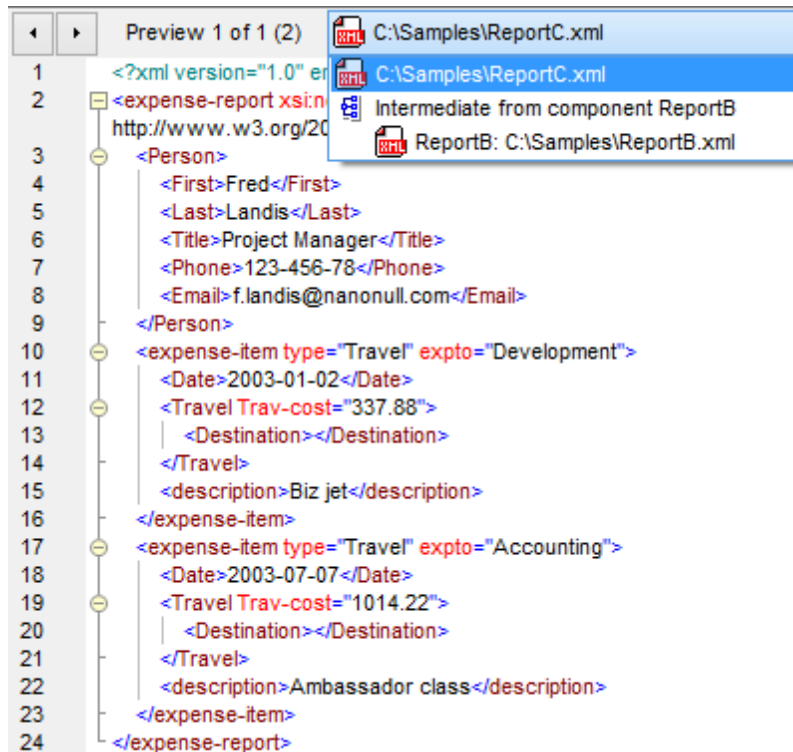
*Settings of the target component*

If you preview the mapping by clicking the **Output** tab in the mapping window, two files are shown in the output, as expected:

1. **ReportB.xml**, which represents the result of the mapping A to B
2. **ReportC.xml**, which represents the result of mapping B to C.

To select which of the two generated output files should be displayed in the Output window, either click the arrow buttons, or select the desired entry from the dropdown list.





*Generated output files*

When the mapping is executed by MapForce, the setting "Write directly to final output file" (configured from **Tools | Options | General**) determines whether the intermediate files are saved as temporary files or as physical files. Note that this is only valid when the mapping is previewed directly in MapForce. Had this mapping been executed by MapForce Server or by generated code, actual files would be produced at each stage in the mapping chain.

If StyleVision is installed, and if a StyleVision Power Stylesheet (SPS) file has been assigned to the target component (as in this example), then the final mapping output can be viewed (and saved as) HTML, RTF file. To generate and view this output in MapForce, click the tab with the corresponding name.





## Personal Expense Report

Currency: ☒ Dollars ☐ Euros ☐ Yen Currency \$

☒ Detailed report

### Employee Information

<input type="text" value="Fred"/>	<input type="text" value="Landis"/>	<input type="text" value="Project Manager"/>
First Name	Last Name	Title
<input type="text" value="f.landis@nanonull.com"/>		<input type="text" value="123-456-78"/>
E-Mail		Phone

### Expense List


Type	Expense To	Date (yyyy-mm-dd)	Expenses \$		Description
<input type="text" value="Travel"/>	<input type="text" value="Development"/>	2003-01-02	<input type="text" value="Travel"/> 337.88	<input type="text" value="Lodging"/>	Biz jet
<input type="text" value="Travel"/>	<input type="text" value="Accounting"/>	2003-07-07	<input type="text" value="Travel"/> 1014.22	<input type="text" value="Lodging"/>	Ambassador class

Mapping DB Query Output ☒ HTML ☐ RTF ☐ PDF ☐ Word 2007+

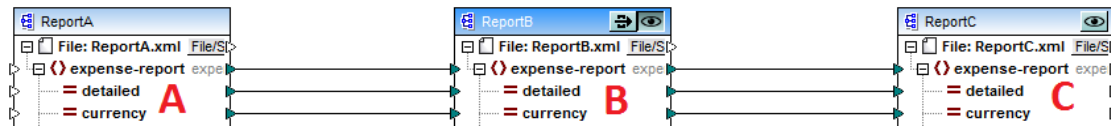
Generated HTML output



Note that only the output of the final target component in the mapping chain is displayed. To display StyleVision output of intermediary components, you would need to deactivate the pass-through button, and preview the intermediate component (as shown in [Example: Pass-Through Inactive](#)).

### 5.3.2 Example: Pass-Through Inactive


The mapping used in this example (**ChainedReports.mfd**) is available in the <Documents> \Altova\MapForce2018\MapForceExamples\Tutorial\ folder. This example illustrates how output is generated differently when the pass-through button  is deactivated on the intermediate component.





As explained in [Example: Pass-Through Active](#), the goal of the mapping is to produce two separate reports. In the previous example, the pass-through button was active , and both reports were generated as expected and could be viewed in the **Output** tab. However, if you want to preview only one of the reports (either **ReportB.xml** or **ReportC.xml**), then the pass-through button must be deactivated () . More precisely, deactivating the pass-through button may be useful if you want to achieve the following:


- Preview only output generated from A to B, and disregard the portion of the mapping from B to C
- Preview only output generated from B to C, and disregard the portion of the mapping from A to B.

When you deactivate the pass-through button as shown above, you can choose whether to preview either **ReportB** or **ReportC** (notice that both have preview  buttons).

Deactivating the pass-through button also lets you to choose what input file should be read by the intermediate component. In most cases, this should be the same file as defined in **Output XML File** field (as in this example).

*Settings of the intermediate component*

Having the same input and output file on the intermediate component is particularly important if you intend to generate code from the mapping, or run the mapping with MapForce Server. As previously mentioned, in these environments, all outputs created by each component in the mapping chain are generated. So, it usually makes sense for the intermediate component to receive one file for processing (in this case **ReportB.xml**) and forward the same file to the subsequent mapping, rather than look for a different file name. Be aware that, not having the same input and output file names on the intermediate component (when the pass-through button is inactive) might cause errors such as "The system cannot find the file specified" in generated code or in MapForce Server execution.

If you click the preview button  on the third component (**ReportC**), and attempt to preview the mapping in MapForce, you will notice that an execution error occurs. This is expected, since,



according to the settings above, a file called **ReportB.xml** is expected as input. However, the mapping did not produce yet such a file (because the pass-through button is not active, and only the portion of the mapping from B to C is executed). You can easily fix this problem as follows:

1. Click the preview button on the intermediate component.
2. Click the **Output** tab to preview the mapping.
3. Save the resulting output file as **ReportB.xml**, in the same folder as the mapping (**<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\**).

Now, if you click again the preview button on the third component (**ReportC**), the error is no longer shown.

When the pass-through button is inactive, you can also preview the StyleVision-generated output for each component that has an associated StyleVision Power StyleSheet (SPS) file. In particular, you can view the HTML version of the intermediate report as well (in addition to that of the final report):





## Personal Expense Report

Currency: ☒ Dollars ☐ Euros ☐ Yen **Currency \$**  
☒ Detailed report

### Employee Information

Fred	Landis	Project Manager
First Name	Last Name	Title
f.landis@nanonull.com		123-456-78
E-Mail		Phone

### Expense List

Type	Expense To	Date (yyyy-mm-dd)	Expenses \$		Description
Travel	Development	2003-01-02	Travel 337.88	Lodging	Biz jet
Travel	Accounting	2003-07-07	Travel 1014.22	Lodging	Ambassador class
Travel	Marketing	2003-02-02	Travel 2000	Lodging	Hong Kong

Mapping DB Query Output **HTML** RTF PDF Word 2007+

HTML output of the intermediate component



## 5.4 Processing Multiple Input or Output Files Dynamically

You can configure MapForce to process multiple files (for example, all files in a directory) when the mapping runs. Using this feature, you can solve tasks such as:

- Supply to the mapping a list of input files to be processed
- Generate as mapping output a list of files instead of a single output file
- Generate a mapping application where both the input and output file names are defined at runtime
- Convert a set of files to another format
- Split a large file into smaller parts
- Merge multiple files into one large file

You can configure a MapForce component to process multiple files in one of the following ways:

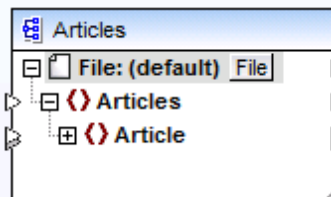
- Supply the path to the required input or output file(s) using wildcard characters instead of a fixed file name, in the component settings (see [Changing the Component Settings](#)). Namely, you can enter the wildcards \* and ? in the Component Settings dialog box, so that MapForce resolves the corresponding path when the mapping runs.
- Connect to the root node of a component a sequence which supplies the path dynamically (for example, the result of the `replace-fileext` function). When the mapping runs, MapForce will read dynamically all the input files or generate dynamically all the output files.

Depending on what you want to achieve, you can use either one or both of these approaches on the same mapping. However, it is not meaningful to use both approaches at the same time on the same component. To instruct MapForce which approach you want to use for a particular component, click the **File** ( [File](#) ) or **File/String** ( [File/String](#) ) button available next to the root node of a component. This button enables you to specify the following behavior:

### *Use File Names from Component Settings*

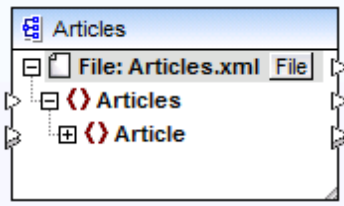
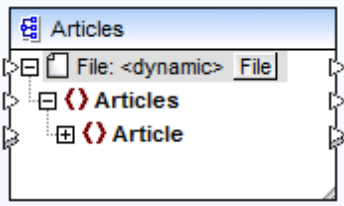
If the component should process one or several instance files, this option instructs MapForce to process the file name(s) defined in the Component Settings dialog box.

If you select this option, the root node does not have an input connector, as it is not meaningful.



If you did not specify yet any input or output files in the Component Settings dialog box, the name of the root node is **File: (default)**. Otherwise, the root node displays the name of the input file, followed by



	<p>a semi-colon ( ;), followed by the name of the output file.</p> <p>If the name of the input is the same with that of the output file, it is displayed as name of the root node.</p>  <p>Note that you can select either this option or the <i>Use Dynamic File Names Supplied by Mapping</i> option.</p>
<p><i>Use Dynamic File Names Supplied by Mapping</i></p>	<p>This option instructs MapForce to process the file name(s) that you define on the mapping area, by connecting values to the root node of the component.</p> <p>If you select this option, the root node gets an input connector to which you can connect values that supply dynamically the file names to be processed during mapping execution. If you have defined file names in the Component Settings dialog box as well, those values are ignored.</p> <p>When this option is selected, the name of the root node is displayed as <b>File: &lt;dynamic&gt;</b>.</p>  <p>This option is mutually exclusive with the <i>Use File Names from Component Settings</i> option.</p>

Multiple input or output files can be defined for the following components:

- XML files
- Text files (CSV\*, FLF\* files and FlexText\*\* files)
- EDI documents\*\*
- Excel spreadsheets\*\*
- XBRL documents\*\*



\* Requires MapForce Professional Edition

\*\* Requires MapForce Enterprise Edition

The following table illustrates support for dynamic input and output file and wildcards in MapForce languages.

Target language	Dynamic input file name	Wildcard support for input file name	Dynamic output file name
XSLT 1.0	*	Not supported by XSLT 1.0	Not supported by XSLT 1.0
XSLT 2.0	*	*(1)	*
C++	*	*	*
C#	*	*	*
Java	*	*	*
BUILT-IN	*	*	*

Legend:

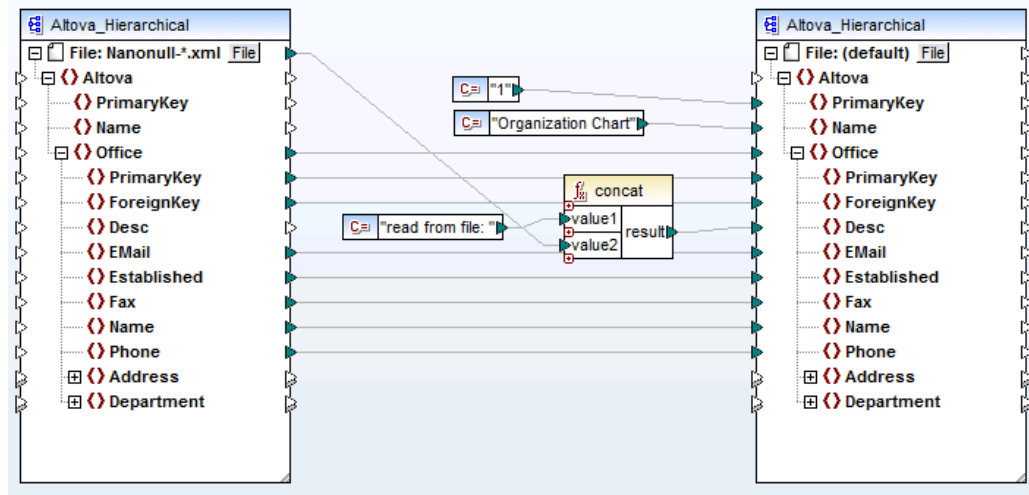
*	Supported
(1)	Uses the <code>fn:collection</code> function. The implementation in the Altova XSLT 2.0 and XQuery engines resolves wildcards. Other engines may behave differently. For details on how to transform XSLT 1.0/2.0 code using the RaptorXML Server engine, see <a href="#">Generating XSLT 1.0, or 2.0 code</a>

### 5.4.1 Mapping Multiple Input Files to a Single Output File

To process multiple input files, do one of the following:

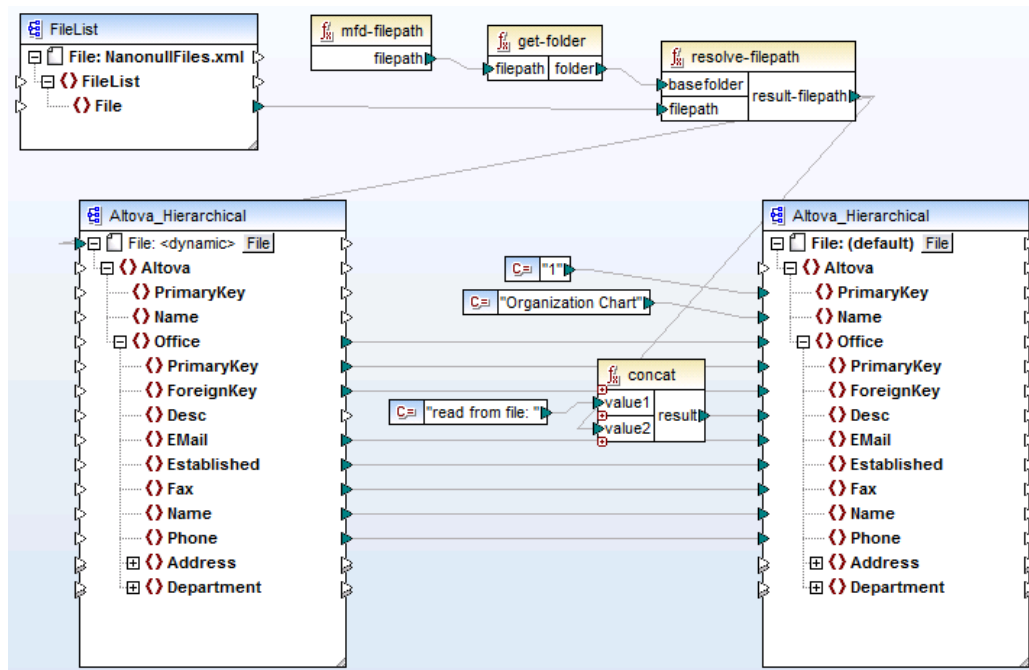
- Enter a file path with wildcards (\* or ?) as input file in the Component Settings dialog box. All matching files will be processed. The example below uses the \* wildcard character in the Input XML file field to supply as mapping input all files whose name begins with "Nanonull-". Multiple input files are being merged into a **single** output file because there is no dynamic connector to the target component, while the source component accesses multiple files using the wildcard \*. Notice that the name of the root node in the target component is **File: <default>**, indicating that no output file path has been defined in the Component Settings dialog box. The multiple source files are thus appended in the target document.





*MergeMultipleFiles.mfd* (MapForce Basic Edition)

- Map a **sequence** of strings to the *File* node of the source component. Each string in the sequence represents one file name. The strings may also contain wildcards, which are automatically resolved. A sequence of file names can be supplied by components such as an XML file.



*MergeMultipleFiles\_List.mfd* (MapForce Basic Edition)

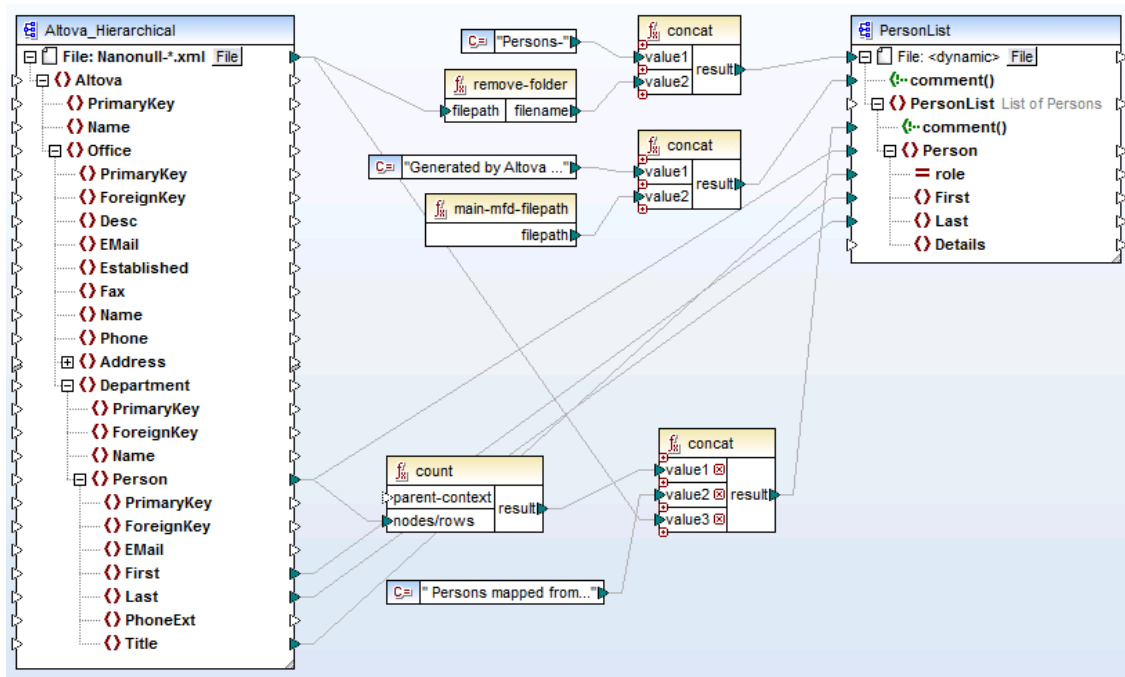
### 5.4.2 Mapping Multiple Input Files to Multiple Output Files

To map multiple files to multiple target files, you need to generate unique output file names. In some cases, the output file names can be derived from strings in the input data, and in other



cases it is useful to derive the output file name from the input file name, e.g. by changing the file extension.

In the following mapping, the output file name is derived from the input file name, by adding the prefix "Persons-" with the help of the **concat** function.



*MultipleInputToMultipleOutputFiles.mfd (MapForce Basic Edition)*

**Note:** Avoid simply connecting the input and output root nodes directly, without using any processing functions. Doing this will overwrite your input files when you run the mapping. You can change the output file names using functions such as the **concat** function, as shown above.

The menu option **File | Mapping Settings** allows you to define globally the file path settings used by the mapping (see [Changing the mapping settings](#)).

### 5.4.3 Supplying File Names as Mapping Parameters

To supply custom file names as input parameters to the mapping, do the following:

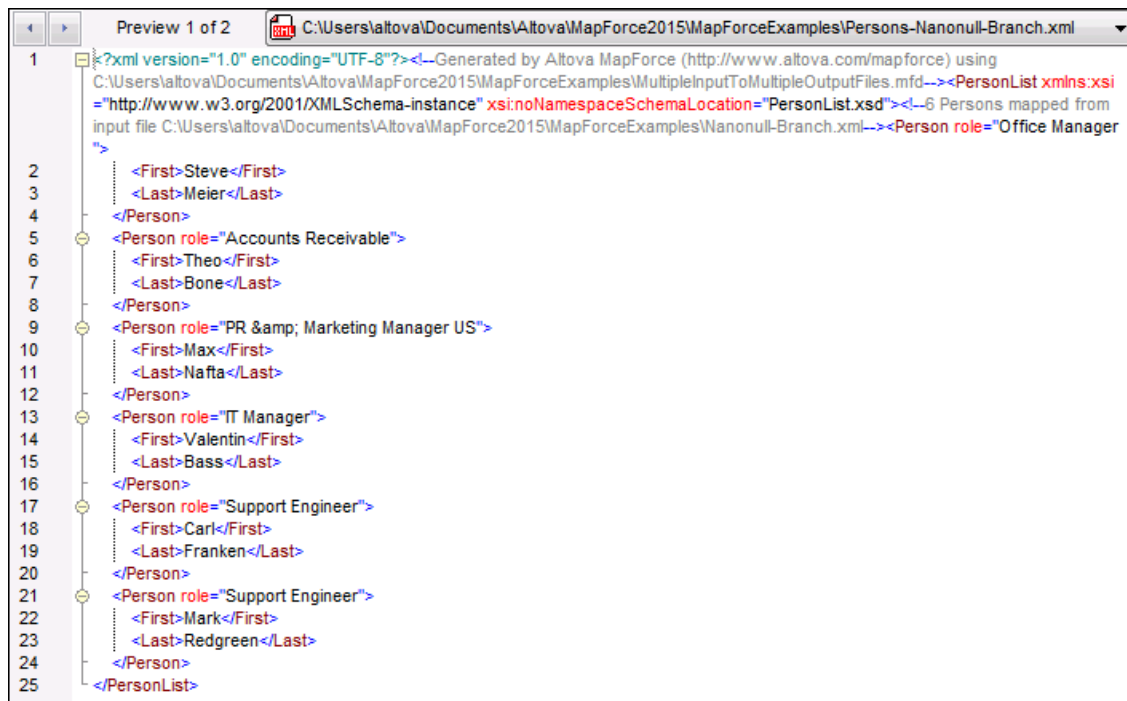
1. Add an Input component to the mapping (On the **Function** menu, click **Insert Input**). For more information about such components, see [Simple Input](#).
1. Click the **File** ( **File** ) or **File/String** ( **File/String** ) button of the source component and select **Use Dynamic File Names Supplied by Mapping**.
2. Connect the Input component to the root node of the component which acts as mapping source.

For a worked example, see [Example: Using File Names as Mapping Parameters](#).



### 5.4.4 Previewing Multiple Output Files

Click the Output tab to display the mapping result in a preview window. If the mapping produces multiple output files, each file has its own numbered pane in the Output tab. Click the arrow buttons to see the individual output files.



*MultipleInputToMultipleOutputFiles.mfd*

To save the generated output files, do one of the following:

- On the **Output** menu, click **Save All Output Files** (  ).
- Click the **Save all generated outputs** (  ) toolbar button.

### 5.4.5 Example: Split One XML File into Many

This example shows you how to generate dynamically multiple XML files from a single source XML file. The accompanying mapping for this example is available at the following path:

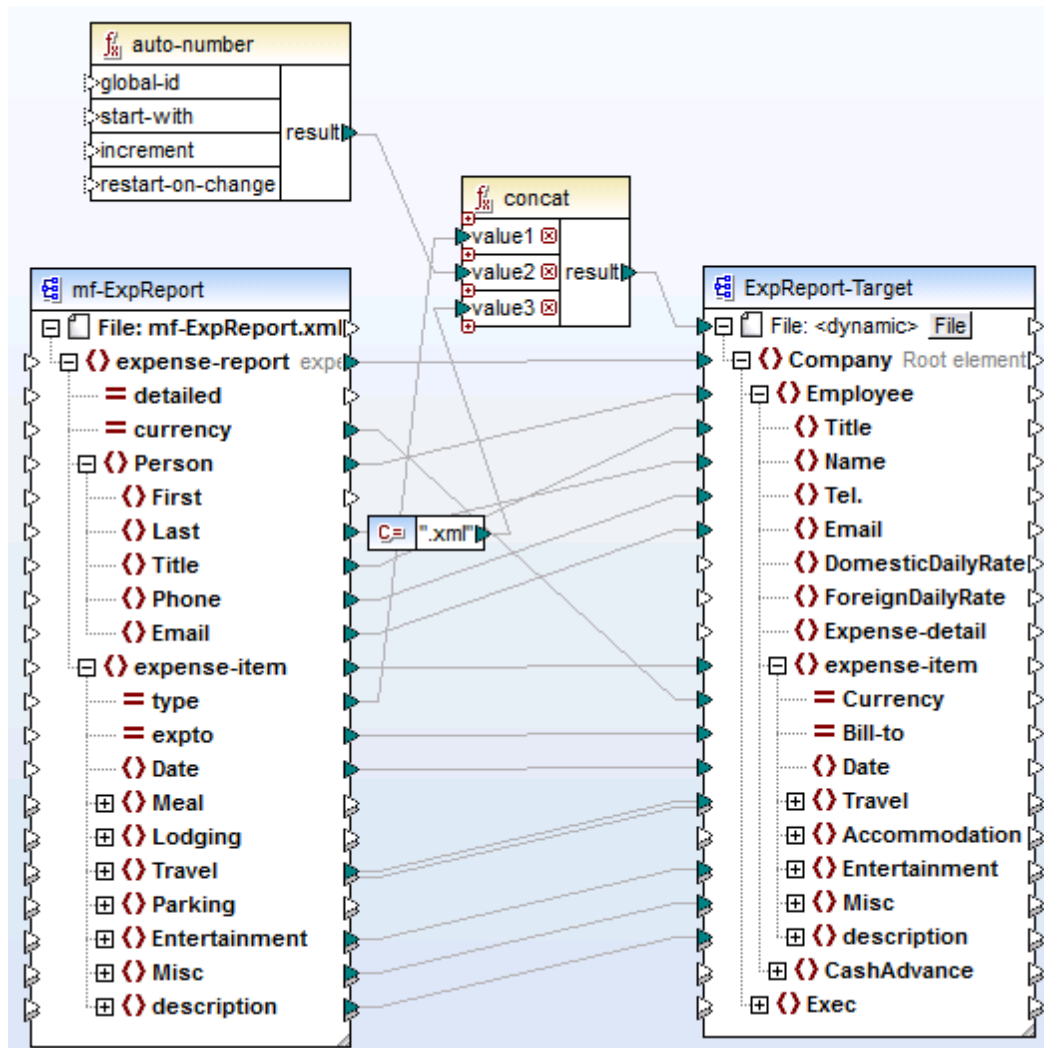
**<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\Tut-ExpReport-dyn.mfd.**

The source XML file (available in the same folder as the mapping) consists of the expense report for a person called "Fred Landis" and contains five expense items of different types. The aim of the example is to generate a separate XML file for each of the expense items listed below.









*Tut-ExpReport-dyn.mfd (MapForce Basic Edition)*

Note that the resulting output files are named dynamically as follows:

- The `type` attribute supplies the first part of the file name (for example, "Travel").
- The `auto-number` function supplies the sequential number of the file (for example, "Travel1", "Travel2", and so on).
- The constant supplies the file extension, which is ".xml", thus "Travel1.xml" is the file name of the first file.



## 5.5 Supplying Parameters to the Mapping

You can pass simple values to a mapping by means of simple input components. On the mapping area, simple input components play the role of a source component which has a simple data type (for example, string, integer, and so on) instead of a structure of items and sequences. Consequently, you can create a simple input component instead of (or in addition to) a file-based source component. In the generated XSLT file, simple input components correspond to stylesheet parameters.

You can create each simple input component (or parameter) as optional or mandatory (see [Input Component Settings](#)). If necessary, you can also create default values for the mapping input parameters (see [Creating a Default Input Value](#)). This enables you to safely run the mapping even if you do not explicitly supply a parameter value at mapping execution time.

Input parameters added on the main mapping area should not be confused with input parameters in user-defined functions (see [User-defined functions](#)). There are some similarities and differences between the two, as follows.

Input parameters on the mapping	Input parameters of user-defined functions
Added from <b>Function   Insert Input</b> menu.	Added from <b>Function   Insert Input</b> menu.
Can have simple data types (string, integer, and so on).	Can have simple as well as complex data types.
Applicable to the entire mapping.	Applicable only in the context of the function in which they were defined.

When you create a reversed mapping (using the menu command **Tools | Create Reversed Mapping**), a simple input component becomes a simple output component.

For an example, see [Example: Using File Names as Mapping Parameters](#).

### 5.5.1 Adding Simple Input Components

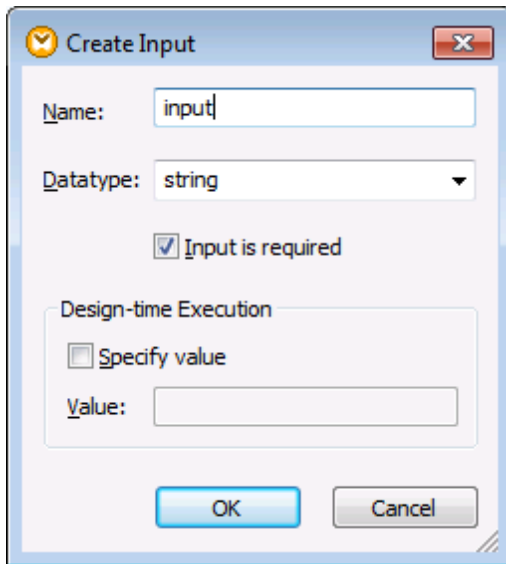
**To add a simple input to the mapping:**

1. Make sure that the mapping window displays the main mapping (not a user-defined function).
2. On the **Function** menu, click **Input**.
3. Enter a name and select the data type required for this input. If the input should be treated as a mandatory mapping parameter, select the **Input is required** check box. For a complete list of settings, see [Simple Input Component Settings](#).

**Note:** The parameter name can contain only letters, digits, and underscores; no other characters are allowed. This makes it possible for a mapping to work across all code generation languages.

4. Click **OK**.



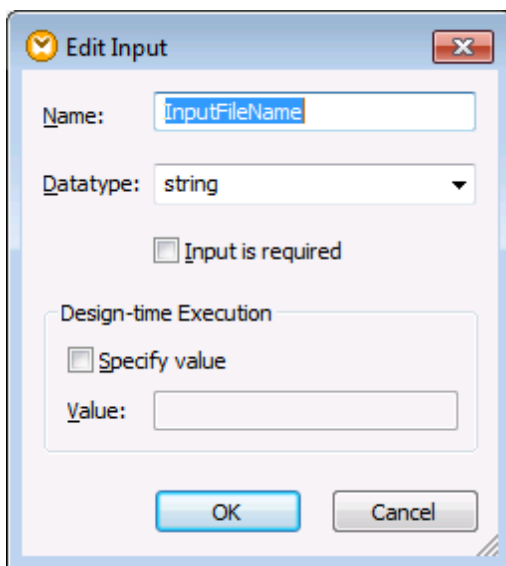


Create Input dialog box

You can change later any of the settings defined here (see [Simple Input Component Settings](#)).

## 5.5.2 Simple Input Component Settings

You can define the settings applicable to a simple input component when adding it to the mapping area. You can also change the settings at a later time, from the Edit Input dialog box.



Edit Input dialog box



To open the **Edit Input** dialog box, do one of the following:

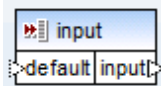
- Select the component, and, on the **Component** menu, click **Properties**.
- Double-click the component.
- Right-click the component, and then click **Properties**.

The available settings are as follows.

<i>Name</i>	Enter a descriptive name for the input parameter corresponding to this component. At mapping execution time, the value entered in this text box becomes the name of the parameter supplied to the mapping; therefore, no spaces or special characters are allowed.
<i>Datatype</i>	By default, any input parameter is treated as string data type. If the parameter should have a different data type, select the respective value from the list. When the mapping is executed, MapForce casts the input parameter to the data type selected here.
<i>Input is required</i>	When enabled, this setting makes the input parameter mandatory (that is, the mapping cannot be executed unless you supply a parameter value).  Disable this check box if you want to specify a default value for the input parameter (see <a href="#">Creating a Default Input Value</a> ).
<i>Specify value</i>	This setting is applicable only if you execute the mapping during design time, by clicking the <b>Preview</b> tab. It allows you to enter directly in the component the value to use as mapping input.
<i>Value</i>	This setting is applicable only if you execute the mapping during design time, by clicking the <b>Preview</b> tab. To enter a value to be used by MapForce as mapping input, select the <b>Specify Value</b> check box, and then type the required value.

### 5.5.3 Creating a Default Input Value

After you add an Input component to the mapping area, notice the **default** item to the left of the component.

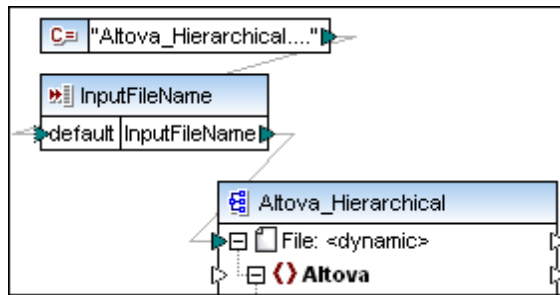


*Simple input component*

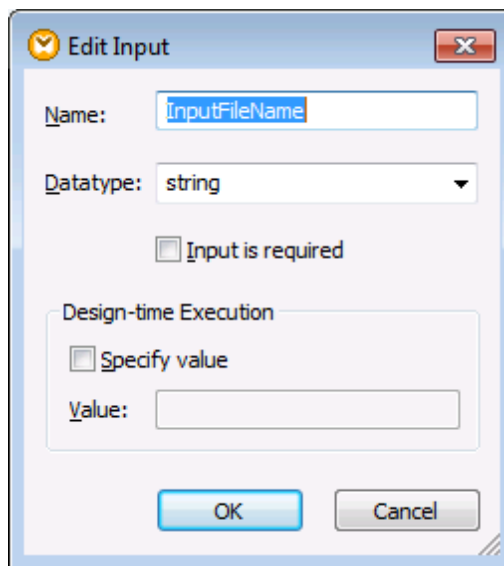
The default item enables you to connect an optional default value to this input component, as follows:

1. Add a constant component (on the **Insert** menu, click **Constant**), and then connect it to the **default** item of the input component.





2. Double click the input component and make sure that the **Input is required** check box is disabled. When you create a default input value, this setting is not meaningful and causes mapping validation warnings.



3. Click **OK**.

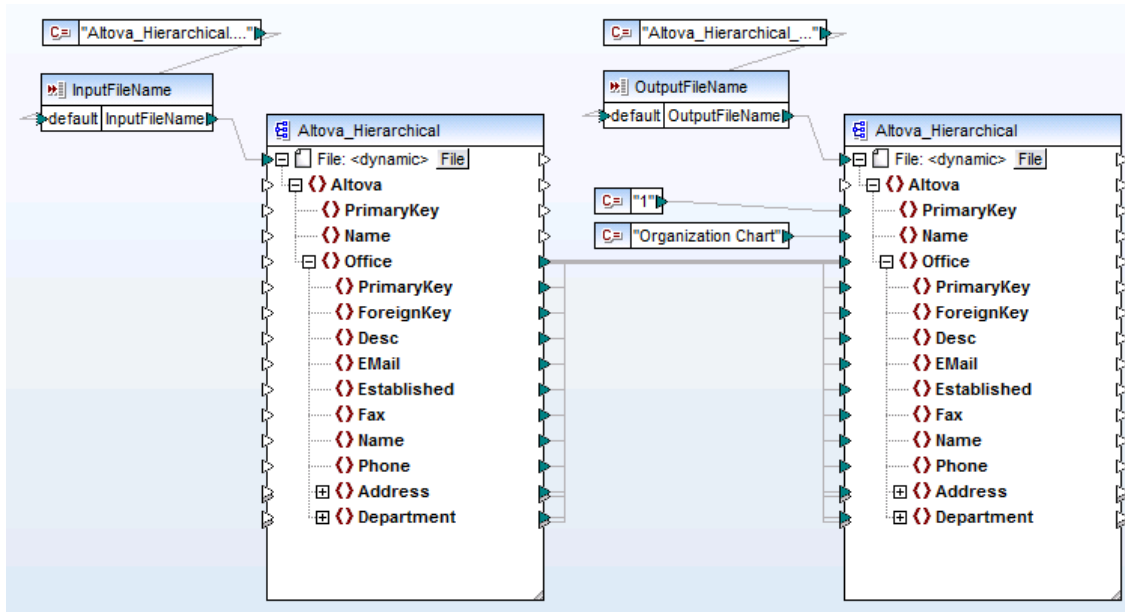
**Note:** If you click the **Specify value** check box and enter a value in the adjacent box, the entered value takes precedence over the default value when you preview the mapping (that is, at design-time execution). However, the same value has no effect in the generated code.

### 5.5.4 Example: Using File Names as Mapping Parameters

This example walks you through the steps required to execute a mapping that takes input parameters at runtime. The mapping design file used in this example is available at the following path: **<Documents>\Altova\MapForce2018\MapForceExamples\FileNameAsParameters.mfd**.

The mapping uses two input components: **InputFileName** and **OutputFileName**. These supply the input file name (and the output file name, respectively) of the source and target XML file. For this reason, they are connected to the **File: <dynamic>** item.





*FileNamesAsParameters.mfd (MapForce Basic Edition)*

Both the **InputFileName** and **OutputFileName** components are simple input components in the mapping, so you can supply them as input parameters when executing the mapping. The following sections illustrate how to do this in the following transformation languages:

- [XSLT 2.0](#), using RaptorXML Server

## XSLT 2.0

If you generate code in XSLT 1.0 or XSLT 2.0, the input parameters are written to the **DoTransform.bat** batch file, for execution by RaptorXML Server (see [Automation with RaptorXML Server](#)). To use a different input (or output) file, you can either pass the required parameters at command line, when calling the **DoTransform.bat** file, or edit the latter to include the required parameters.

To supply a custom input parameter in the **DoTransform.bat** file:

1. Generate the XSLT 2.0 code (**File | Generate Code In | XSLT 2.0**) from the **FileNamesAsParameters.mfd** sample.
2. Copy the **Altova\_Hierarchical.xml** file from the **<Documents>\Altova\MapForce2018\MapForceExamples\** directory to the directory where you generated the XSLT 2.0 code (in this example, **c:\codegen\examples\xslt2\**). This file will act as custom parameter.
3. Edit **DoTransform.bat** to include the custom input parameter either before or after **%\*** (as highlighted below). Note that the parameter value is enclosed with single quotes. The available input parameters are listed in the **rem** (Remark) section.

```
@echo off

RaptorXML xslt --xslt-version=2 --
```



```
input="MappingMapToAltova_Hierarchical.xslt" --  
param=InputFileName:'Altova_Hierarchical.xml' %*  
"MappingMapToAltova_Hierarchical.xslt"  
rem --param=InputFileName:  
rem --param=OutputFileName:  
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
```

When you run the DoTransform.bat file, RaptorXML Server completes the transformation using **Altova\_Hierarchical.xml** as input parameter.



```
C:\codegen\examples\xslt2>DoTransform.bat  
file:///C:/codegen/examples/xslt2/MappingMapToAltova_Hierarchical.xslt: result="OK"  
xslt-main-output-files="" xslt-additional-output-files="file:///C:/codegen/examples/xslt2/Altova_Hierarchical_output.xml"
```




## 5.6 Returning String Values from a Mapping

Use a simple output component when you need to return a string value from the mapping. On the mapping area, simple output components play the role of a target component which has a string data type instead of a structure of items and sequences. Consequently, you can create a simple output component instead of (or in addition to) a file-based target component. For example, you can use a simple output component to quickly test and preview the output of a function (see [Example: Testing Function Output](#)).

Simple output components should not be confused with output parameters of user-defined functions (see [User-defined functions](#)). There are some similarities and differences between the two, as follows.

Output components	Output parameters of user-defined functions
Added from <b>Function   Insert Output</b> menu.	Added from <b>Function   Insert Output</b> menu.
Have "string" as data type.	Can have simple as well as complex data types.
Applicable to the entire mapping.	Applicable only in the context of the function in which they were defined.

If necessary, you can add multiple simple output components to a mapping. You can also use simple output components in combination with file-based target components. When your mapping contains multiple target components, you can preview the data returned by a particular component by clicking the **Preview** (  ) button in the component title bar, and then clicking the **Output** tab on the Mapping window.

You can use simple output components as follows in MapForce transformation languages:

Language	How it works
XSLT 1.0, XSLT 2.0	<p>If the generated XSLT files, a simple output components defined in the mapping becomes the output of the XSLT transformation.</p> <p>If you are using RaptorXML Server, you can instruct RaptorXML Server to write the mapping output to the file passed as value to the <code>--output</code> parameter.</p> <p>To write the output to a file, add or edit to the <code>--output</code> parameter in the <b>DoTransform.bat</b> file. For example, the following <b>DoTransform.bat</b> file has been edited to write the mapping output to the <b>Output.txt</b> file (see highlighted text).</p> <pre>RaptorXML xslt --xslt-version=2 -- input="MappingMapToResult1.xslt" -- output="Output.txt" %* "MappingMapToResult1.xslt"</pre>



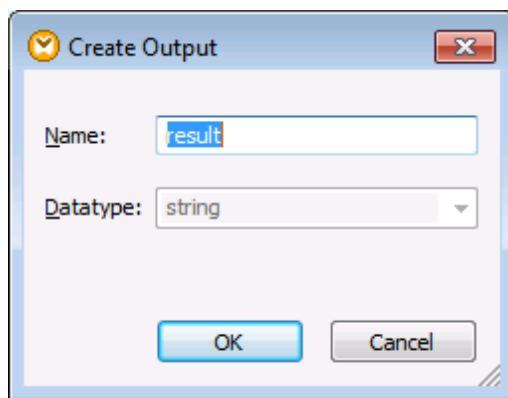
	If an <code>--output</code> parameter is not defined, the mapping output will be written to the standard output stream (stdout) when the mapping is executed.
--	---------------------------------------------------------------------------------------------------------------------------------------------------------------

When you create a reversed mapping (using the menu command **Tools | Create Reversed Mapping**), the simple output component becomes a simple input component.

### 5.6.1 Adding Simple Output Components

To add an Output component to the mapping area:

1. Make sure that the mapping window displays the main mapping (not a user-defined function).
2. On the **Function** menu, click **Output**.
3. Enter a name for the component.
4. Click **OK**.



Create Output dialog box

You can change the component name at any time later, in one of the following ways:

- Select the component, and, on the **Component** menu, click **Properties**.
- Double-click the component header.
- Right-click the component header, and then click **Properties**.

### 5.6.2 Example: Previewing Function Output

This example illustrates how to preview the output returned by MapForce functions with the help of simple output components. You will make the most of this example if you already have a basic understanding of functions in general, and of MapForce functions in particular. If you are new to MapForce functions, you may want to refer to [Using Functions](#) before continuing.

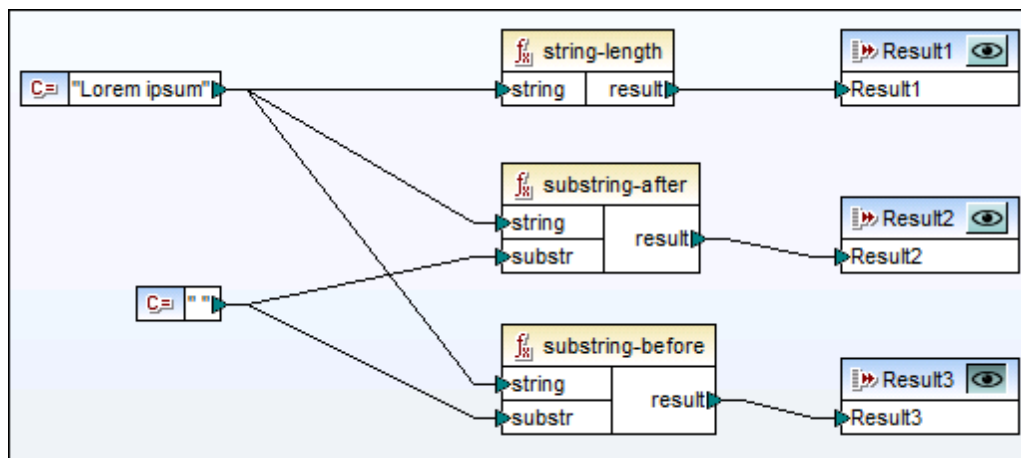
Our aim is to add a number of functions to the mapping area, and learn how to preview their output with the help of simple output components. In particular, the example uses a few simple functions available in the core library. Here is a summary of their usage:



<u><a href="#">string-length</a></u>	Returns the number of characters in the string provided as argument. For example, if you pass to this function the value "Lorem ipsum", the result is "11", since this is the number of characters that the text "Lorem ipsum" takes.
<u><a href="#">substring-after</a></u>	Returns the part of the string that occurs after the separator provided as argument. For example, if you pass to this function the value "Lorem ipsum" and the space character (" "), the result is "ipsum".
<u><a href="#">substring-before</a></u>	Returns the part of the string that occurs before the separator provided as argument. For example, if you pass to this function the value "Lorem ipsum" and the space character (" "), the result is "Lorem".

To test each of these functions against a custom text value ("Lorem ipsum", in this example), follow the steps below:

1. Add a constant with the value "Lorem ipsum" to the mapping area (use the menu command **Insert | Constant**). The constant will be the input parameter for each of the functions to be tested.
2. Add the **string-length**, **substring-after**, and **substring-before** functions to the mapping area, by dragging them to the mapping area from the core library, **string functions** section.
3. Add a constant with an empty space (" ") as value. This will be the separator parameter required by the **substring-after** and **substring-before** functions.
4. Add three simple output components (use the menu command **Function | Insert Output**). In this example, they have been named *Result1*, *Result2*, and *Result3*, although you can give them another title.
5. Connect the components as illustrated below.




*Testing function output with simple output components*

As shown in the sample above, the "Lorem ipsum" string acts as input parameter to each of the **string-length**, **substring-after**, and **substring-before** functions. In addition to this, the **substring-after** and **substring-before** functions take a space value as second input parameter. The **Result1**, **Result2**, and **Result3** components can be used to preview the result of each function.



**To preview the output of any function**

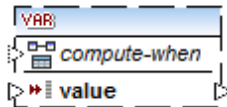
- Click the **Preview** (  ) button in the component title bar, and then click the **Output** tab on the Mapping window.



## 5.7 Using Variables

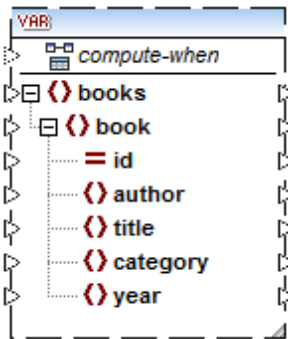
Variables are a special type of component used to store an intermediate mapping result for further processing. They might be necessary in situations where you want to temporarily "remember" some data on the mapping and process it in some way (for example, filter it, or apply some functions) before it is copied to the target component.

Variables can be of simple type (for example, string, integer, boolean, etc) or complex type (a tree structure).



*Simple variable*

You can create a variable of complex type by supplying an XML schema which expresses the structure of the variable. If the schema defines any elements globally, you can choose which one should become the root node of the variable structure. Note that a variable does not have any associated instance XML file; the data of the variable is computed at mapping runtime.

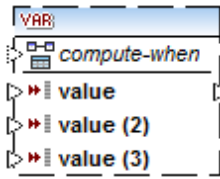


*Complex variable created from an XML schema*

In the images above, you may notice that each variable has an item called `compute-when`. Connecting this item is optional; this enables you to control how the variable value should be computed on the mapping (see [Changing the Context and Scope of Variables](#)).

When necessary, items of a variable structure can be duplicated to accept data from more than one source connection, similar to how this is done for standard components (see [Duplicating Input](#)). This does not apply, however, to variables created from database tables.






Simple variable with duplicated inputs

One of the most important things about variables is that they are sequences, and can be used to create sequences. The term "sequence" here means a list of zero or more items (see also [Mapping Rules and Strategies](#)). This makes it possible for a variable to process multiple items for the duration of the mapping lifetime. If, however, you want to assign a value once to a variable and keep it the same for the rest of the mapping, it is also possible (see [Changing the Context and Scope of Variables](#)).

To some extent, variables can be compared to intermediate components of a chained mapping (see [Chained Mappings](#)). However, they are more flexible and convenient if you don't need to produce intermediary files at each stage in the mapping. The following table outlines differences between variables and chained mappings.


Chained mappings	Variables
Chained mappings involve two totally independent steps. For example, let's assume a mapping that has three components A, B, and C. Running the mapping involves two stages: executing the mapping from A to B, and then executing the mapping from B to C.	While the mapping is executed, variables are evaluated according to their context and scope. Their context and scope can be influenced (see <a href="#">Changing the Context and Scope of Variables</a> ).
When the mapping is executed, intermediate results are stored externally in files.	When the mapping is executed, intermediate results are stored internally. No external files containing a variable's results are produced.
The intermediate result can be previewed using the preview  button.	A variable's result cannot be previewed, since it is computed at mapping runtime.

**Note:** Variables are not supported if the mapping transformation language is set to XSLT 1.0.

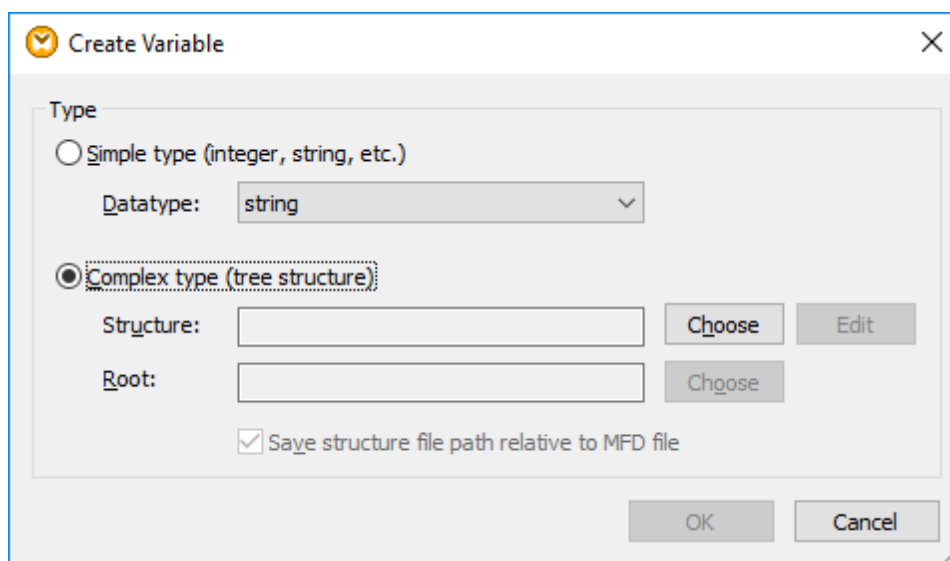
### 5.7.1 Adding Variables

There are several ways to add variables to a mapping, as shown below.

#### Using a menu or toolbar command

1. On the **Insert** menu, click **Variable**. (Alternatively, click the **Variable**  toolbar button).



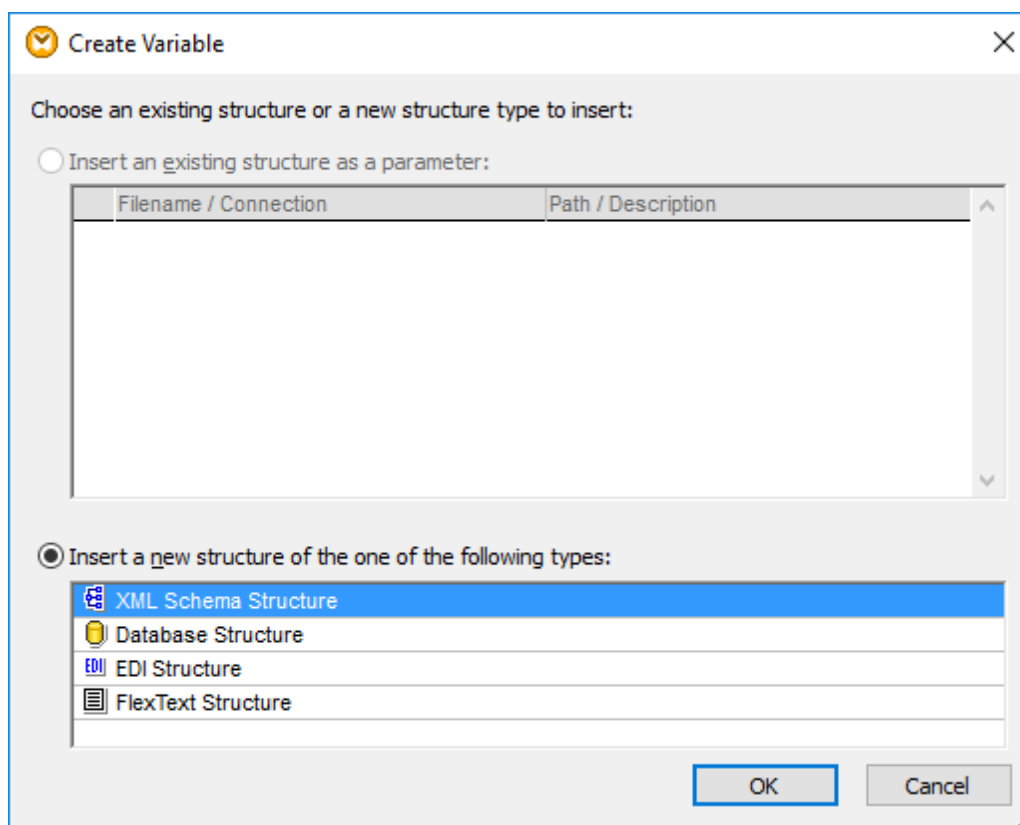


The "Create Variable" dialog box is shown. It has a title bar with a yellow arrow icon and a close button. The "Type" section has two radio buttons: "Simple type (integer, string, etc.)" and "Complex type (tree structure)". The "Simple type" option is currently selected. Below it, the "Datatype:" dropdown menu is set to "string". The "Complex type (tree structure)" option is also visible. Below it, there are two text input fields: "Structure:" and "Root:". To the right of the "Structure:" field are "Choose" and "Edit" buttons. To the right of the "Root:" field is a "Choose" button. At the bottom of the "Complex type" section, there is a checked checkbox labeled "Save structure file path relative to MFD file". At the bottom right of the dialog are "OK" and "Cancel" buttons.

2. Select the type of variable you want to insert (simple or complex type).

If you select "Complex type", there are a few additional steps:

3. Click **Choose** to select the source which should provide the structure of the variable (for example, an XML Schema ).

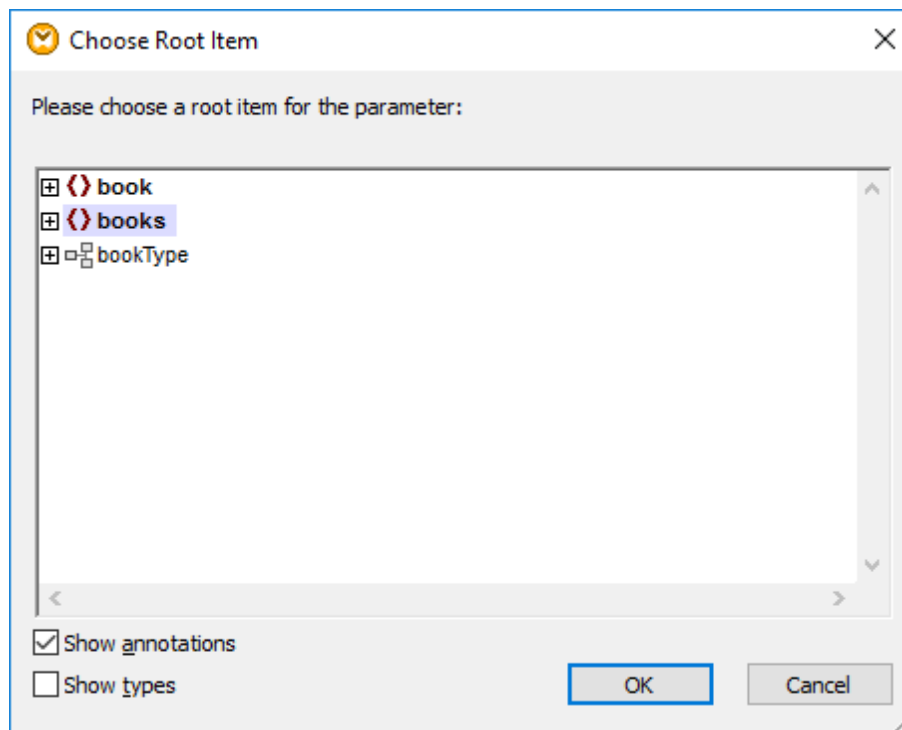


The "Create Variable" dialog box is shown, now with the "Complex type (tree structure)" option selected. The "Type" section is expanded. The "Structure:" field is empty, and the "Choose" button is highlighted. Below the "Structure:" field, there is a list box titled "Choose an existing structure or a new structure type to insert:". The list box has two columns: "Filename / Connection" and "Path / Description". The list box is currently empty. Below the list box, there is a radio button labeled "Insert a new structure of the one of the following types:". This radio button is selected. Below it, there is a list box with four options: "XML Schema Structure", "Database Structure", "EDI Structure", and "FlexText Structure". The "XML Schema Structure" option is selected. At the bottom right of the dialog are "OK" and "Cancel" buttons.

4. When prompted, specify a root item of the structure. In case of XML Schemas, the root

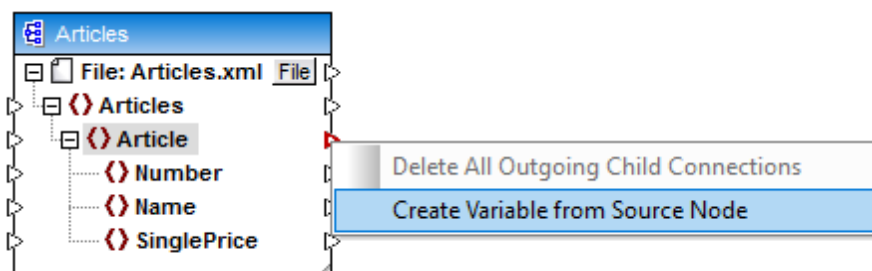


item can be any element defined globally. In case of databases, the root item can be any table.



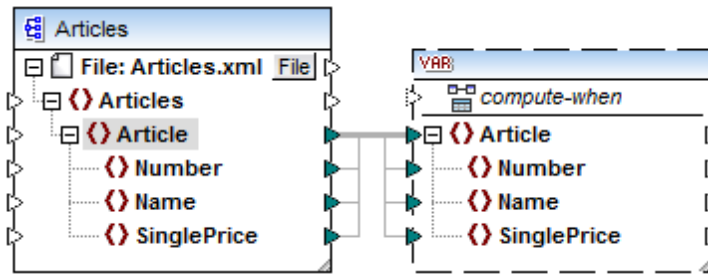
### Using a context menu

- Right-click the output connector of a component (in this example, "Article") and select **Create Variable from Source node**.



This creates a complex variable using the same source schema and automatically connects all items with a Copy-All connection.

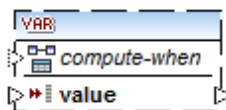




- Right-click the input connector of a target component and select **Create Variable for Target Node**. This creates a complex variable using the same schema as the target, and automatically connects all items with a Copy-All connection.
- Right-click the output connector of a filter component (on-true/on-false) and select **Create Variable from Source Node**. This creates a complex component using the source schema, and automatically uses the item linked to the filter input as the root element of the intermediate component.

## 5.7.2 Changing the Context and Scope of Variables

Every variable has a `compute-when` input item which allows you to control the scope of the variable; in other words, when and how often the variable value is computed when the mapping is executed. You do not have to connect this input in many cases, but it can be essential to override the default context, or to optimize the mapping performance.



The "compute-when" item

In the following text, a *subtree* means the set of an item/node in a target component and all of its descendants, for example, a `<Person>` element with its `<FirstName>` and `<LastName>` child elements.

A *variable value* means the data that is available at the output side of the variable component.

- For simple variables, it is a sequence of atomic values that have the datatype specified in the component properties.
- For complex variables, it is a sequence of root nodes (of the type specified in the component properties), each one including all its descendant nodes.

The sequence of atomic values (or nodes) may contain one or even zero elements. This depends on what is connected to the input side of the variable, and to any parent items in the source and target components.

### "Compute-when" is not connected (default)

If the `compute-when` input item is not connected (to an output node of a source component), the



variable value is computed *whenever it is first used in a target subtree* (either directly via a connector from the variable component to a node in the target component, or indirectly via functions). The same variable value is also used for all target child nodes inside the subtree.

The actual variable value depends on any connections between parent items of the source and target components.

This default behavior is the same as that of complex outputs of [regular user-defined functions](#) and Web service function calls.

If the variable output is connected to multiple unrelated target nodes, the variable value is computed *separately for each of them*. This can produce different results in each case, because different parent connections influence the context in which the variable's value is evaluated.

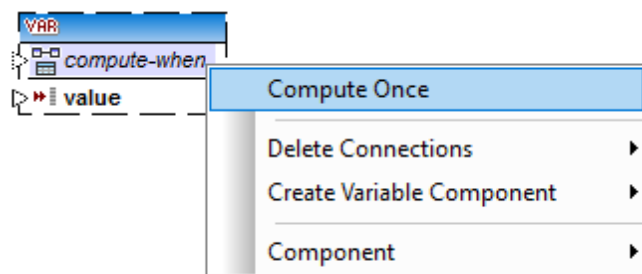
### "Compute-when" is connected

By connecting an output connector of a source component to `compute-when`, the variable is computed *whenever that source item is first used in a target subtree*.

The variable actually acts as if it were a child item of the item connected to `compute-when`. This makes it possible to bind the variable to a specific source item. That is, at runtime the variable is re-evaluated whenever a new item is read from the sequence in the source component. This relates to the general rule governing connections in MapForce: "for each source item, create one target item". With `compute-when`, it means "for each source item, compute the variable value" (see [Mapping Rules and Strategies](#)).

### "Compute-once"

If necessary, you can choose to compute the variable value *once before each of the target components*, making the variable essentially a global constant for the rest of the mapping. To do this, right-click the `compute-when` item and select **Compute Once** from the context menu:



When you change the scope of a variable to `compute-when=once`, the input connector is removed from the `compute-when` item, since such a variable is only evaluated once.

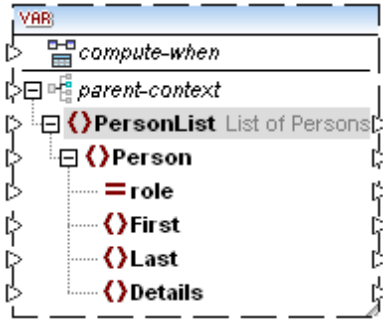
In a user-defined function, the `compute-when=once` variable is evaluated each time the function is called, before the actual function result is evaluated.



## Parent-context

Adding a parent-context may be necessary, for example, if your mapping uses multiple filters and you need an additional parent node to iterate over, see also [Overriding the Mapping Context](#).

To add a parent-context to a variable, right-click the root node (in this example, "PersonList") and select **Add Parent Context** from the context menu. This adds a new node, `parent-context`, to the existing hierarchy.



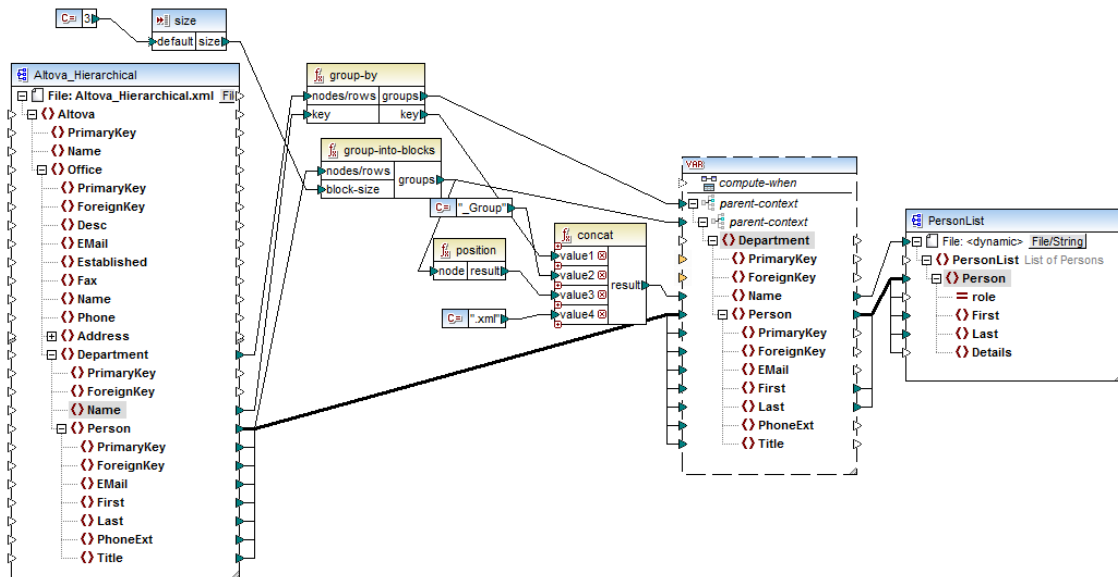
The parent context adds a virtual "parent" node to the hierarchy within the component. This allows you to iterate over an additional node in the same, or in a different source component.

### 5.7.3 Example: Grouping and Subgrouping Records

The mapping illustrated in this example is available as **DividePersonsByDepartmentIntoGroups.mfd** in the `<Documents>\Altova\MapForce2018\MapForceExamples\` folder.

This mapping processes an XML file that contains employee records of a fictitious company. The company has two offices: "Nanonull, Inc." and "Nanonull Partners, Inc". Each office has several departments (for example, "IT", "Marketing", and so on), and each department has one or more employees. The goal of the mapping is to create groups of maximum three people from each department, regardless of the office. The size of each group is three by default; however, it should be easy to change if necessary. Each group must be saved as a separate XML file, with the name having the format "<Department Name>\_GroupN" (for example, **Marketing\_Group1.xml**, **Marketing\_Group2.xml**, and so on).





*DividePersonsByDepartmentIntoGroups.mfd*

As illustrated above, in order to achieve the mapping goal, a complex variable was added to the mapping, and a few other component types (primarily functions). The variable has the same structure as a `Department` item in the source XML. If you right-click the variable in order to view its properties, you will notice that it uses the same XML schema as the source component, and has `Department` as root element. Importantly, the variable has two nested `parent-context` items, which ensure that the variable is computed first in the context of each department, and then in the context of each group within each department (see also [Changing the Context and Scope of Variables](#)).

Initially, the mapping iterates through all departments in order to obtain the name of each department (this will be subsequently required to create the file name corresponding to each group). This is achieved by connecting the [group-by](#) function to the `Department` source item, and by supplying the department name as grouping key.

Next, within the context of each department, a second grouping takes place. Namely, the mapping calls the [group-into-blocks](#) function in order to create the required groups of people. The size of each group is supplied by a simple input component which has a default value of "3". The default value is supplied by a constant. In this example, in order to change the size of each group, one can easily modify the constant value as required. However, the "size" input component can also be modified so that, if the mapping is run by generated code or with MapForce Server, the size of each group could be conveniently supplied as a parameter to the mapping. For more information, see [Supplying Parameters to the Mapping](#).

Next, the value of the variable is supplied to the target `PersonList` XML component. The file name for each created group was computed by concatenating the following parts, with the help of the [concat](#) function:

1. The name of each department
2. The string `"_Group"`
3. The number of the group in the current sequence (for example, "1" if this is the first group)



- for this department)
4. The string ".xml"

The result of this concatenation is stored in the `Name` item of the variable, and then supplied as a dynamic file name to the target component. This causes a new file name to be created for each received value. In this example, the variable computes eight groups in total, so eight output files are created when the mapping runs, as required. For more information about this technique, see [Processing Multiple Input or Output Files Dynamically](#).

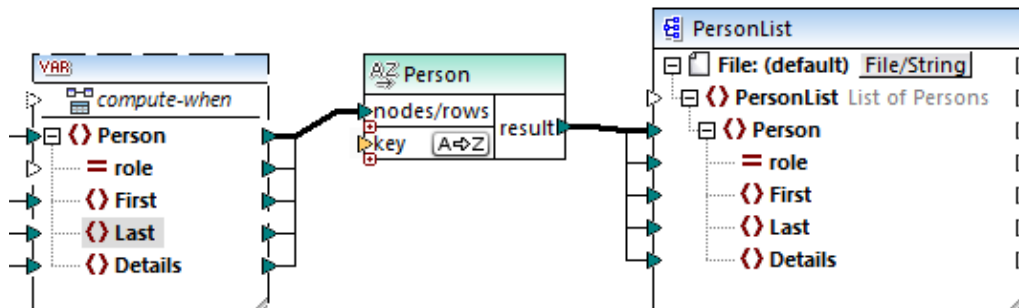



## 5.8 Sorting Data

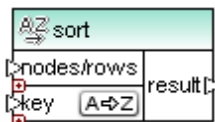
To sort input data based on a specific sort key, use a Sort component. The Sort component supports the following target languages: XSLT2, XQuery, and Built-in.

**To add a sort component to the mapping, do one of the following:**

- Right-click an existing connection, and select **Insert Sort: Nodes/Rows** from the context menu. This inserts the Sort component and automatically connects it to the source and target components. For example, in the mapping below, the Sort component was inserted between a variable and an XML component. The only thing that remains to be connected manually is the sorting key (the field by which you want to sort).



- On the **Insert** menu, click **Sort** (alternatively, click the **Sort**  toolbar button). This inserts the Sort component in its "unconnected" form.

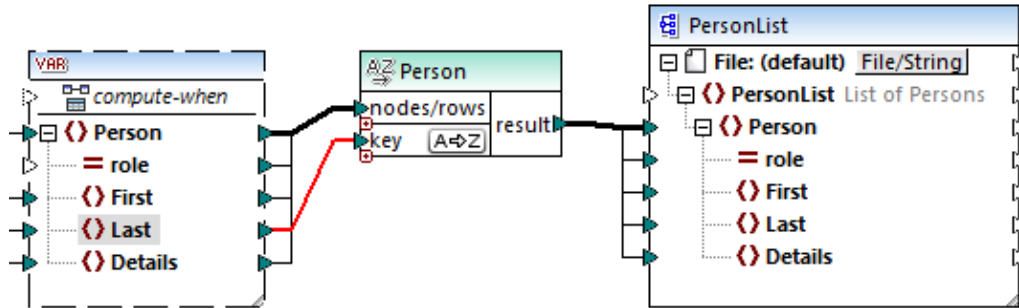


As soon as a connection is made to the source component, the title bar name changes to that of the item connected to the `nodes/rows` item.

**To define the item by which you want to sort:**

- Connect the item by which you want to sort to the `key` parameter of the Sort component. For example, in the mapping below, the `Person` nodes/rows are sorted by the field `Last`.



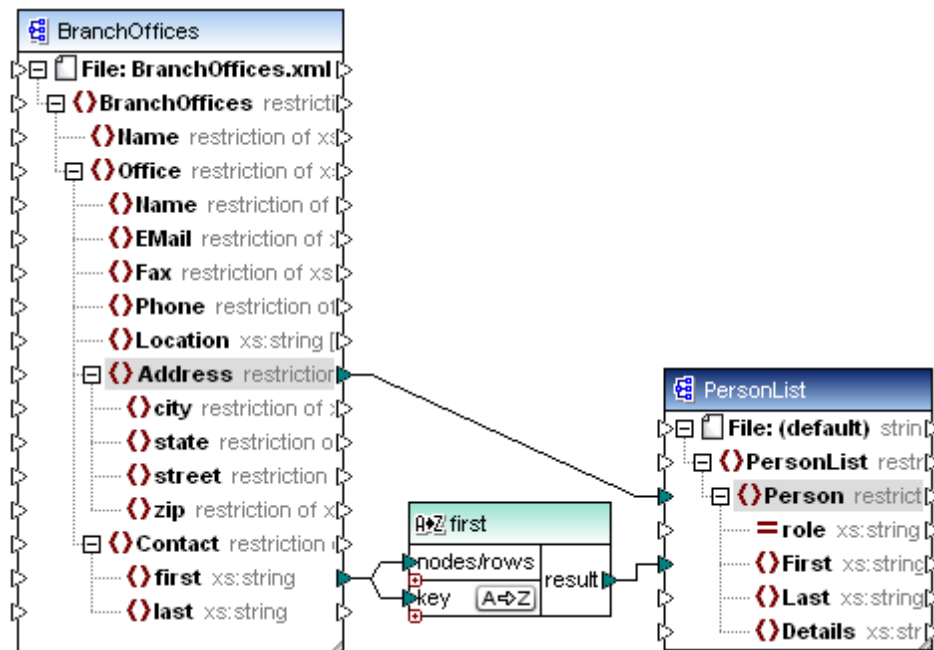


To change the sort order:

- Click the **A↔Z** icon in the Sort component. It changes to **Z↔A** to show that the sort order has been reversed.

To sort input data consisting of simple type items:

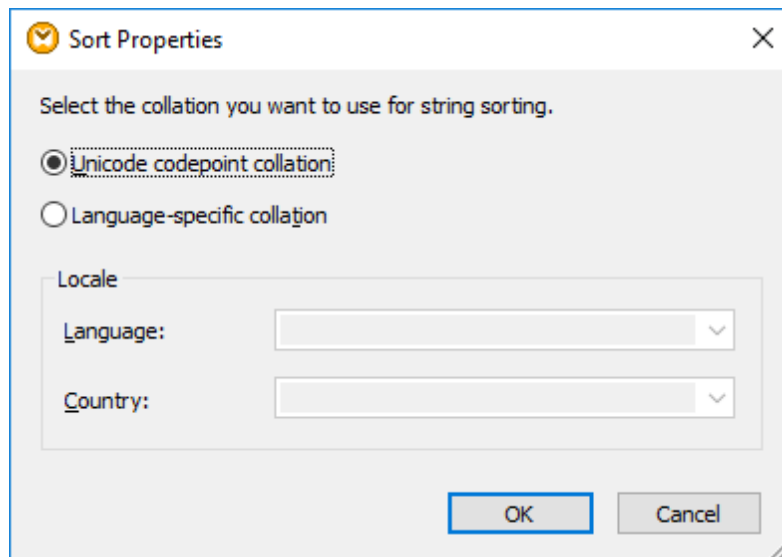
- Connect the item to both the `nodes/rows` and `key` parameters of the sort component. In the mapping below, the element of simple type `first` is being sorted.



To sort strings using language-specific rules:

- Double-click the header of the Sort component to open the Sort Properties dialog box.



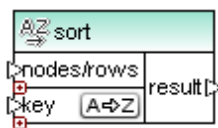


**Unicode codepoint collation:** This (default) option compares/orders strings based on code point values. Code point values are integers that have been assigned to abstract characters in the Universal Character Set adopted by the Unicode Consortium. This option allows sorting across many languages and scripts.

**Language-specific collation:** This option allows you to define the specific language and country variant you want to sort by. This option is supported when using the BUILT-IN execution engine. For XSLT, support depends on the specific engine used to execute the code.




### 5.8.1 Sorting by Multiple Keys

After you add a Sort component to the mapping, one sorting key called `key` is created by default.



*Default Sort component*

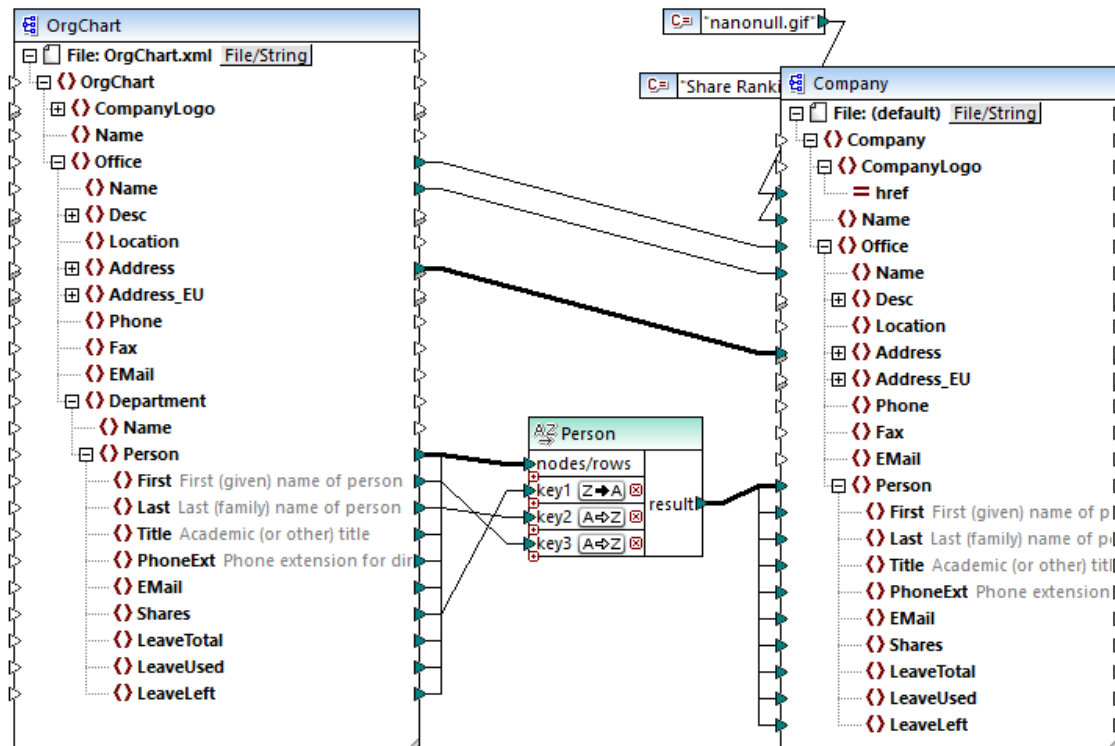
If you want to sort by multiple keys, adjust the Sort component as follows:

- Click the **Add Key** (  ) icon to add a new key (for example, `key2` in the mapping below).
- Click the **Delete Key** (  ) icon to delete a key.
- Drop a connection onto the  icon to add a key and also connect to it.

A mapping which illustrates sorting by multiple key is available at the following path:

**<Documents>\Altova\MapForce2018\MapForceExamples\SortByMultipleKeys.mfd.**





SortByMultipleKeys.mfd

In the mapping above, *Person* records are sorted by three sorting keys:

1. *Shares* (number of shares a person holds)
2. *Last* (last name)
3. *First* (first name)

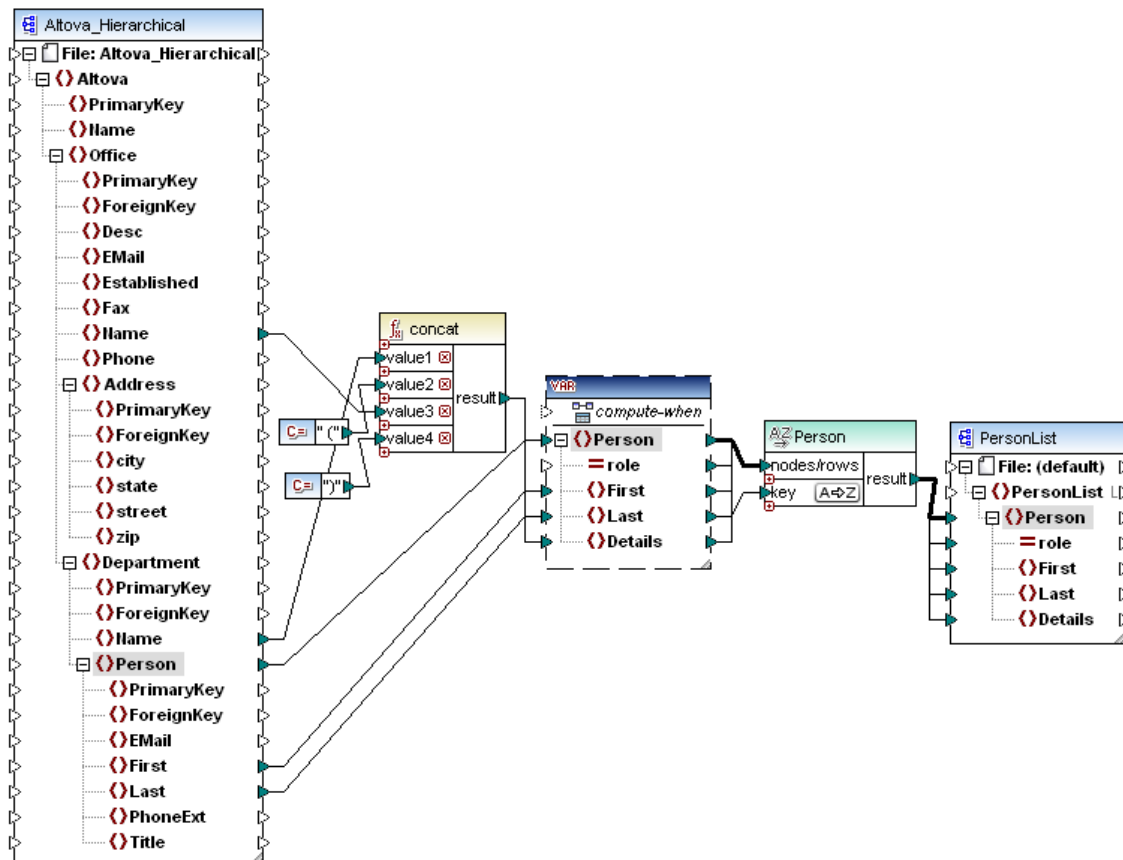
Note that the position of the sorting key in the Sort component determines its sort priority. For example, in the mapping above, records are initially sorted by the number of shares. This is the sorting key with the highest priority. If the number of shares is the same, people are then sorted by their last name. Finally, when multiple people have the same number of shares and the same last name, the person's first name is taken into account.

The sort order of each key can be different. In the mapping above, the key *Shares* has a descending sort order (Z-A), while the other two keys have ascending sort order (A-Z).

## 5.8.2 Sorting with Variables

In some cases, it may be necessary to add intermediate variables to the mapping in order to achieve the desired result. This example illustrates how to extract records from an XML file, and sort them, with the help of intermediate variables. The example is accompanied by a mapping sample located at the following path: **<Documents>\Altova\MapForce2018\MapForceExamples\Altova\_Hierarchical\_Sort.mfd**.





Altova\_Hierarchical\_Sort.mfd

This mapping reads data from a source XML file called **Altova\_Hierarchical.xml** and writes it to a target XML file. As shown above, the source XML contains information about a fictitious company. The company is divided into offices. Offices are sub-divided into departments, and departments are further divided into people.

The target XML component, **PersonList**, contains a list of **Person** records. The **Details** item is meant to store information about the office and department where the person belongs.

The aim is to extract all persons from the source XML and sort them alphabetically by last name. Also, the office and department name where each person belongs must be written to the **Details** item.

To achieve this goal, this example makes use of the following component types:

1. The **concat** function. In this mapping, this function returns a string in the format `Office(Department)`. It takes as input the office name, the department name, and two constants which supply the start and end brackets. See also [Working with Functions](#).
2. An intermediate variable. The role of the variable is to bring all data relevant to a person into the same mapping context. The variable causes the mapping to look up the department and office of each person, in the context of each person. To put it differently, the variable "remembers" the office and department name to which a person belongs. Without the variable, the context would be incorrect, and the mapping would produce





unwanted output (for more information about how a mapping is executed, see [Mapping Rules and Strategies](#)). Notice that the variable replicates the structure of the target XML file (it uses the same XML schema). This makes it possible to connect the sort result to the target, through a Copy-All connection. See also [Using Variables](#) and [Copy-All Connections](#).

3. A Sort component, which performs the actual sorting. Notice that the key input of the `Sort` component is connected to the `Last` item of the variable, which sorts all person records by their last name.



## 5.9 Filters and Conditions

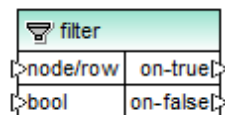
When you need to filter data, or get a value conditionally, you can use one of the following component types:

- Filter: Nodes/Rows (  )
- If-Else Condition (  )

You can add these components to the mapping either from the **Insert** menu, or from the **Insert Component** toolbar. Importantly, each of the components above has specific behavior and requirements. The differences are explained in the following sections.

### Filtering nodes or rows

When you need to filter data, including XML nodes, use a **Filter Nodes/Rows** component. The **Filter Nodes/Rows** component enables you to retrieve a subset of nodes from a larger set of data, based on a true or false condition. Its structure on the mapping area reflects this:



In the structure above, the condition connected to **bool** determines whether the connected **node/row** goes to the **on-true** or **on-false** output. Namely, if the condition is true, the **node/row** will be redirected to the **on-true** output. Conversely, if the condition is false, the **node/row** will be redirected to the **on-false** output.

When your mapping needs to consume only items that *meet* the filter condition, you can leave the **on-false** output unconnected. If you need to process the items that *do not meet* the filter condition, connect the **on-false** output to a target where such items should be redirected.

For a step-by-step mapping example, see [Example: Filtering Nodes](#).

### Returning a value conditionally

If you need to get a single value (not a node or row) conditionally, use an **If-Else Condition**. Note that If-Else conditions are not suitable for filtering nodes or rows. Unlike **Filter Nodes/Rows** components, an **If-Else Condition** returns a value of simple type (such as a string or integer). Therefore, **If-Else Conditions** are only suitable for scenarios where you need to process a simple value conditionally. For example, let's assume you have a list of average temperatures per month, in the format:

```

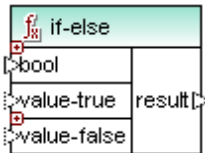
<Temperatures>
  <data temp="19.2" month="2010-06" />
  <data temp="22.3" month="2010-07" />
  <data temp="19.5" month="2010-08" />
  <data temp="14.2" month="2010-09" />
  
```





```
<data temp="7.8" month="2010-10" />
<data temp="6.9" month="2010-11" />
<data temp="-1.0" month="2010-12" />
</Temperatures>
```

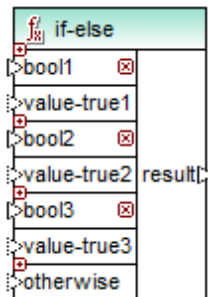
An **If-Else Condition** would enable you to return, for each item in the list, the value "high" if temperature exceeds 20 degrees Celsius, and value "low" if temperature is lower than 5 degrees Celsius.

On the mapping, the structure of the **If-Else Condition** looks as follows:



If the condition connected to **bool** is true, then the value connected to **value-true** is output as **result**. If the condition is false, the value connected to **value-false** is output as **result**. The data type of **result** is not known in advance; it depends on the data type of the value connected to **value-true** or **value-false**. The important thing is that it should always be a simple type (string, integer, and so on). Connecting input values of complex type (such as nodes or rows) is not supported by **If-Else Conditions**.


If-Else Conditions are extendable. This means that you can add multiple conditions to the component, by clicking the **Add** (  ) button. To delete a previously added condition, click the **Delete** (  ) the button. This feature enables you to check for multiple conditions and return a different value for each condition, if it is true.



Expanded **If-Else Conditions** are evaluated from top to bottom (first conditions is checked first, then the second one, and so on). If you want to return a value when none of the conditions are true, connect it to **otherwise**.

For a step-by-step mapping example, see [Example: Returning a Value Conditionally](#).

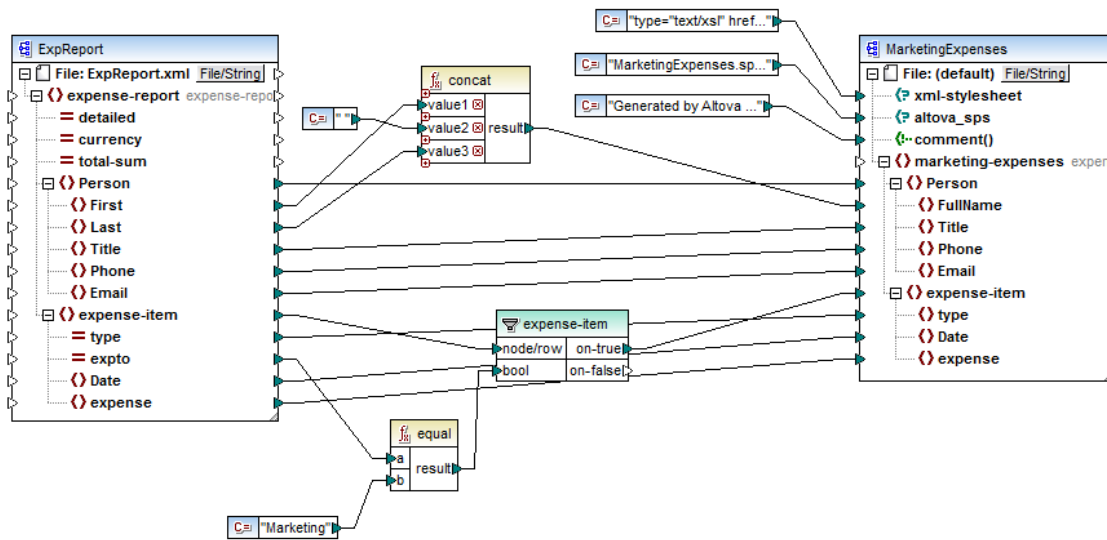
### 5.9.1 Example: Filtering Nodes


This example shows you how to filter nodes based on a true/false condition. A **Filter: Nodes/Rows** (  ) component is used to achieve this goal.

The mapping described in this example is available at the following path: **<Documents>\Altova**



### \\MapForce2018\\MapForceExamples\\MarketingExpenses.mfd.



As shown above, the mapping reads data from a source XML which contains an expense report ("ExpReport") and writes data to a target XML ("MarketingExpenses"). There are several other components between the target and source. The most relevant component is the **expense-item** filter (  ), which represents the subject of this topic.

The goal of the mapping is to filter out only those expense items that belong to the Marketing department. To achieve this goal, a filter component has been added to the mapping. (To add a filter, click the **Insert** menu, and then click **Filter: Nodes/Rows**.)

To identify whether each expense item belongs to Marketing, this mapping looks at the value of the "expto" attribute in the source. This attribute has the value "Marketing" whenever the expense is a marketing expense. For example, in the code listing below, the first and third expense item belongs to Marketing, the second belongs to Development, and the fourth belongs to Sales:

```
...
<expense-item type="Meal" expto="Marketing">
  <Date>2003-01-01</Date>
  <expense>122.11</expense>
</expense-item>
<expense-item type="Lodging" expto="Development">
  <Date>2003-01-02</Date>
  <expense>122.12</expense>
</expense-item>
<expense-item type="Lodging" expto="Marketing">
  <Date>2003-01-02</Date>
  <expense>299.45</expense>
</expense-item>
<expense-item type="Entertainment" expto="Sales">
  <Date>2003-01-02</Date>
  <expense>13.22</expense>
</expense-item>
...
```



*XML input before the mapping is executed*

On the mapping area, the **node/row** input of the filter is connected to the **expense-item** node in the source component. This ensures that the filter component gets the list of nodes that it must process.

To add the condition based on which filtering should occur, we have added the **equal** function from the MapForce core library (for more information, see [Working with Functions](#)). The **equal** function compares the value of the "type" attribute to a constant which has the value "Marketing". (To add a constant, click the **Insert** menu, and then click **Constant**.)


Since we need to filter only those items that satisfy the condition, we connected only the **on-true** output of the filter to the target component.

When you preview the mapping result, by clicking the **Output** tab, MapForce evaluates, for each expense-item node, the condition connected to the **bool** input of the filter. When the condition is true, the expense-item node is passed on to the target; otherwise, it is ignored. Consequently, only the expense items matching the criteria are displayed in the output:

```
...
<expense-item>
  <type>Meal</type>
  <Date>2003-01-01</Date>
  <expense>122.11</expense>
</expense-item>
<expense-item>
  <type>Lodging</type>
  <Date>2003-01-02</Date>
  <expense>299.45</expense>
</expense-item>
...
```

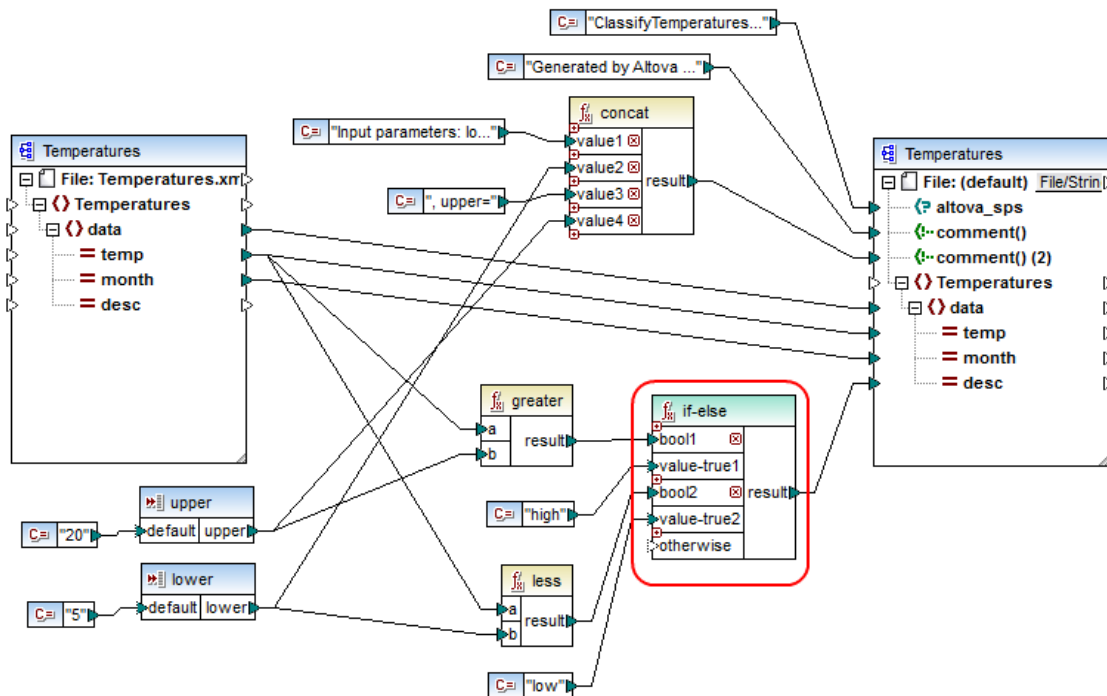
*XML output after the mapping is executed*

## 5.9.2 Example: Returning a Value Conditionally

This example shows you how to return a simple value from a component, based on a true/false condition. An **If-Else Condition** (  ) is used to achieve the goal. Note that **If-Else Conditions** should not be confused with filter components. **If-Else Conditions** are only suitable when you need to process simple values conditionally (string, integer, etc.). If you need to filter complex values such as nodes, use a filter instead (see [Example: Filtering Nodes](#)).


The mapping described in this example is available at the following path: **<Documents>\Altova\MapForce2018\MapForceExamples\ClassifyTemperatures.mfd**.





This mapping reads data from a source XML which contains temperature data ("Temperatures") and writes data to a target XML which conforms to the same schema. There are several other components between the target and source, one of them being the **if-else** condition (highlighted in red), which is also the subject of this topic.

The goal of the mapping is to add short description to each temperature record in the target. Specifically, if temperature is above 20 degrees Celsius, the description should be "high". If the temperature is below 5 degrees Celsius, the description should be "low". For all other cases, no description should be written.

To achieve this goal, conditional processing is required; therefore, an If-Else Condition has been added to the mapping. (To add an If-Else Condition, click the **Insert** menu, and then click **If-Else Condition**.) In this mapping, the If-Else Condition has been extended (with the help of the  button) to accept two conditions: **bool1** and **bool2**.

The conditions themselves are supplied by the **greater** and **less** functions, which have been added from the MapForce core library (for more information, see [Working with Functions](#)). These functions evaluate the values provided by two input components, called "upper" and "lower". (To add an input component, click the **Insert** menu, and then click **Insert Input**. For more information about input components, see [Supplying Parameters to the Mapping](#).)

The **greater** and **less** functions return either true or false. The function result determines what is written to the target instance. Namely, if the value of the "temp" attribute in the source is greater than 20, the constant value "high" is passed to the **if-else** condition. If the value of the "temp" attribute in the source is less than 5, the constant value "low" is passed on to the **if-else** condition. The **otherwise** input is not connected. Therefore, if none of the above conditions is met, nothing is passed to the **result** output connector.

Finally, the **result** output connector supplies this value (once for each temperature record) to the



"desc" attribute in the target.

When you are ready to preview the mapping result, click the **Output** tab. Notice that the resulting XML output now includes the "desc" attribute, whenever the temperature is either greater than 20 or lower than 5.

```
...  
<data temp="-3.6" month="2006-01" desc="low"/>  
<data temp="-0.7" month="2006-02" desc="low"/>  
<data temp="7.5" month="2006-03"/>  
<data temp="12.4" month="2006-04"/>  
<data temp="16.2" month="2006-05"/>  
<data temp="19" month="2006-06"/>  
<data temp="22.7" month="2006-07" desc="high"/>  
<data temp="23.2" month="2006-08" desc="high"/>  
...
```

*XML output after the mapping is executed*



## 5.10 Using Value-Maps

The Value-Map component allows you to transform an input value to a different output value using a lookup table. This is useful for converting different enumeration types. The component only has one input and output item.

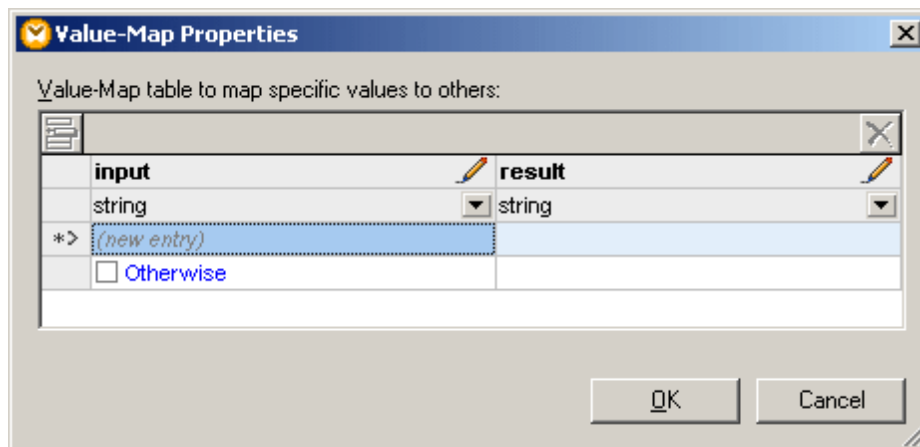
Note: if you want to retrieve/filter data based on specific criteria, please use the **Filter** component, see [Filters and Conditions](#).

To use a Value-Map component:

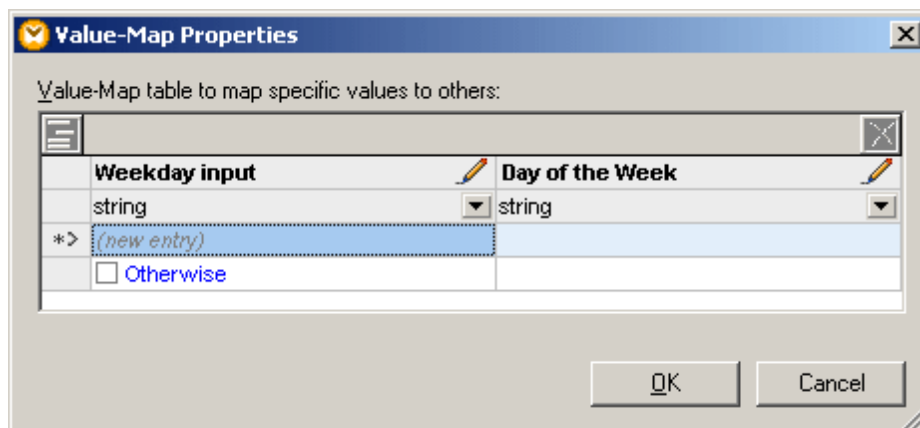
1. Select the menu option **Insert | Value-Map**, or click the Value-Map icon  in the icon bar.



2. Double click the Value-Map component to open the value map table.



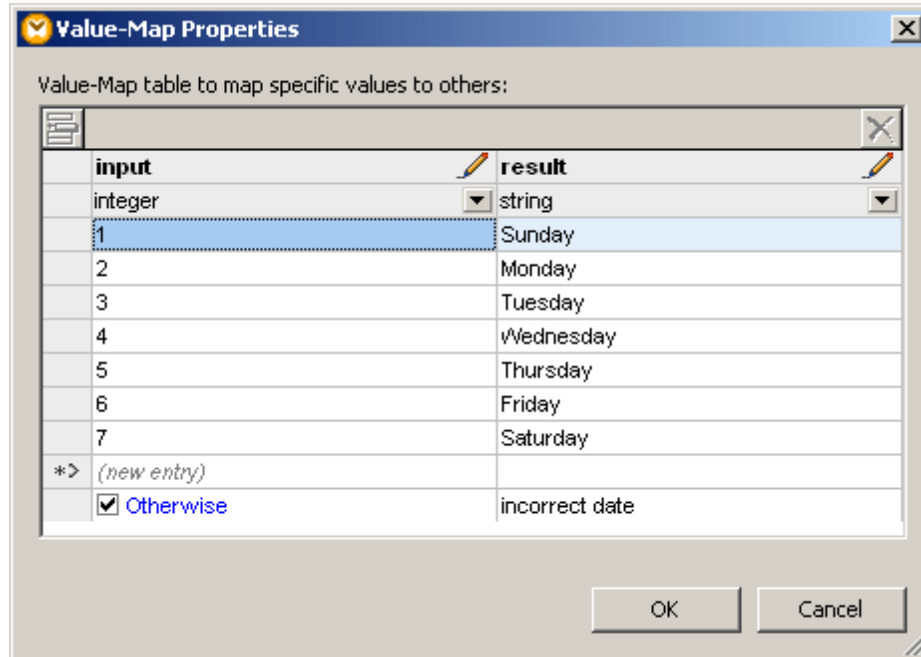
3. Click into the column headers and enter **Weekday input** in the first, and **Day of the Week** in the second.



4. Enter the input value that you want to transform, in the **Weekday input** column.
5. Enter the output value you want to transform that value to, in the **Day of the week**



- column.
6. Simply type in the **(new entry)** input field to enter a new value pair.
  7. Click the **datatype** combo box, below the column header to select the input and output datatypes, e.g. integer and string.

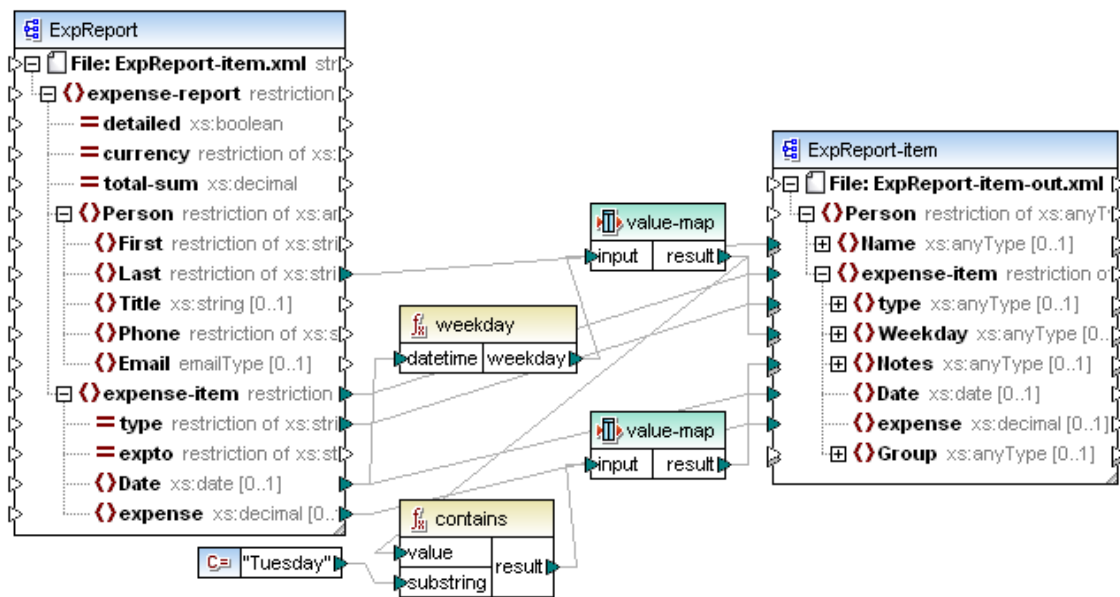


Note: activate the **Otherwise** check box, and enter the value, to define an alternative output value if the supplied values are not available on input. To pass through source data without changing it please see [Passing data through a Value-Map unchanged](#).

8. You can click the edit icons in the header rows to change the column names, which are also displayed in the mapping. This will make it easier to identify the purpose of the component in the mapping.

The **Expense-valmap.mfd** file in the [...\MapForceExamples\Tutorial\](#) folder is a sample mapping that shows how the Value-Map can be used.





What this mapping does:

Extracts the day of the week from the Date item in the data source, converts the numerical value into text, and places it in the Weekday item of the target component i.e. Sunday, Monday etc.

- The **weekday** function extracts the weekday number from the **Date** item in the ExpReport source file. The result of this function are integers ranging from 1 to 7.
- The Value-Map component transforms the integers into weekdays, i.e. Sunday, Monday, etc. as shown in the graphic at the top of this section.
- If the output contains "Tuesday", then the corresponding output "Prepare Financial Reports" is mapped to the Notes item in the target component.
- Clicking the Output tab displays the target XML file with the transformed data.

```

3      <Name>Landis</Name>
4      <expense-item>
5          <type>Meal</type>
6          <Weekday>Tuesday</Weekday>
7          <Notes>-- Prepare financial reports --</Notes>
8          <Date>2003-01-01</Date>
9          <expense>122.11</expense>
10     </expense-item>
11     <expense-item>
12         <type>Lodging</type>
13         <Weekday>Monday</Weekday>
14         <Notes>--</Notes>
15         <Date>2003-01-14</Date>
16         <expense>122.12</expense>
17     </expense-item>

```

Note:

Placing the mouse cursor over the value map component opens a popup containing the currently defined values.

The output from various types of logical, or string functions, can only be a boolean **"true"**

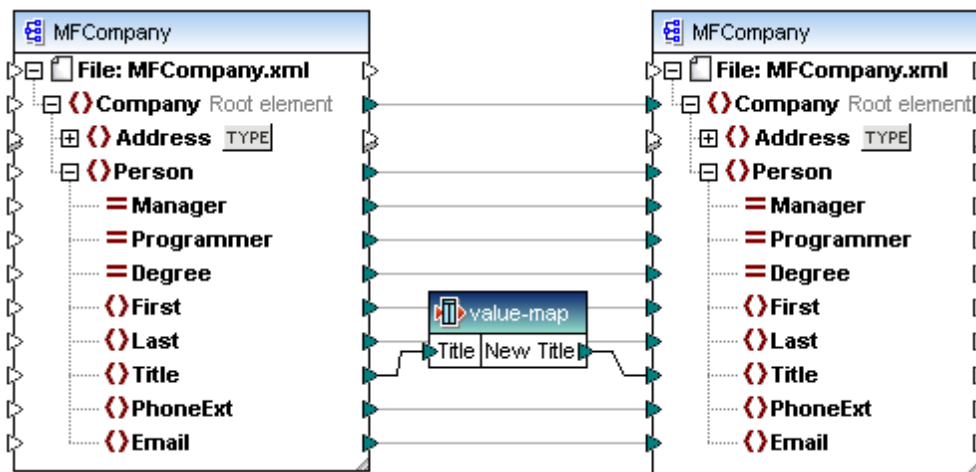


or "false" value. The value you want to test for, must thus be entered into the **input** field of the value map table e.g. "true".

### 5.10.1 Passing data through a Value-Map unchanged

This section describes a mapping situation where some specific node data have to be transformed, while the rest of the node data have to be passed on to the target node unchanged.

An example of this would be a company that changes some of the titles in a subsidiary. In this case it might change two title designations and want to keep the rest as they currently are.



The obvious mapping would be the one shown above, which uses the value-map component to transform the specific titles.

Clicking the Output tab shows us the result of the mapping:

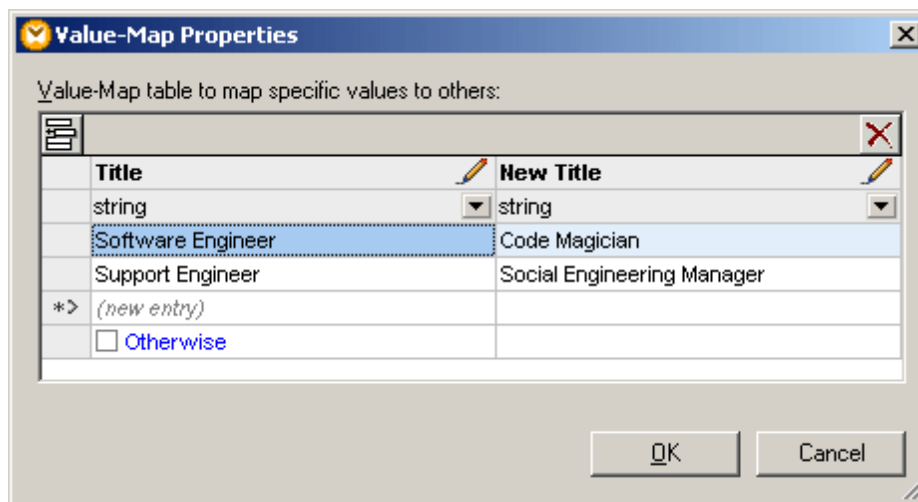
```

33  <Person>
34    <First>Fred</First>
35    <Last>Landis</Last>
36    <PhoneExt>951</PhoneExt>
37    <Email>f.landis@nanonull.com</Email>
38  </Person>
39  <Person>
40    <First>Michelle</First>
41    <Last>Butler</Last>
42    <Title>Code Magician</Title>
43    <PhoneExt>654</PhoneExt>
44    <Email>m.landis@nanonull.com</Email>
45  </Person>

```

For those persons who are neither of the two types shown in the value-map component, the Title element is deleted in the output file.





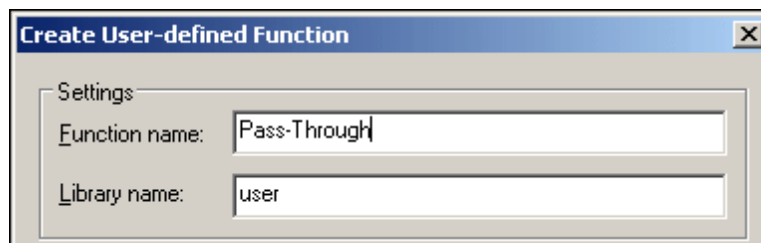
Possible alternative:

Clicking the **Otherwise** check box and entering a substitute term, does make the Title node reappear in the output file, but it now contains the same **New Title** for all other persons of the company.

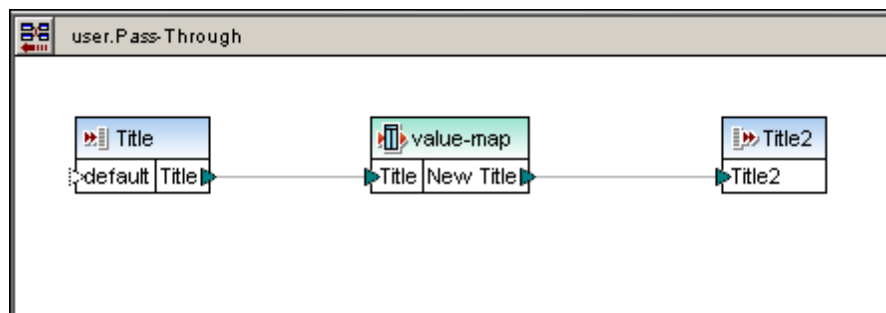
### Solution:

Create a user-defined function containing the value-map component, and use the **substitute-missing** function to supply the original data for the empty nodes.

1. Click the value-map component and select **Function | Create user-defined function from Selection**.

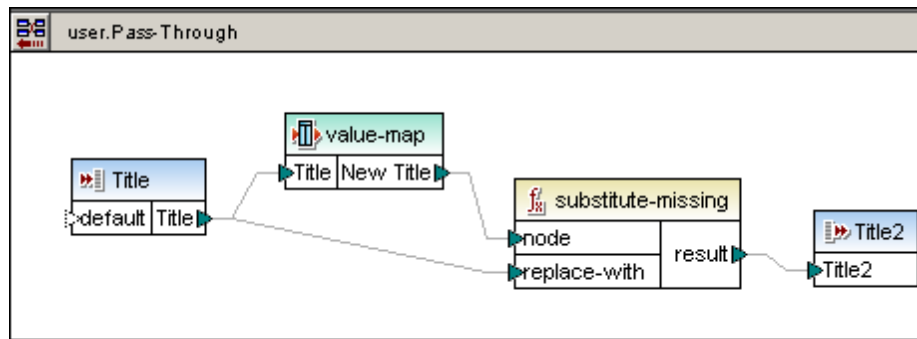


2. Enter a name for the function e.g. Pass-Through and click OK.



3. Insert a **substitute-missing** function from the **core | node function** section of the Libraries pane, and create the connections as shown in the screen shot below.





4. Click the Output tab to see the result:

Result of the mapping:

- The two Title designations in the value-map component are transformed to New Title.
- All other Title nodes of the source file, retain their original Title data in the target file.

```

38  <Person>
39    <First>Fred</First>
40    <Last>Landis</Last>
41    <Title>Program Manager</Title>
42    <PhoneExt>951</PhoneExt>
43    <Email>f.landis@nanonull.com</Email>
44  </Person>
45  <Person>
46    <First>Michelle</First>
47    <Last>Butler</Last>
48    <Title>Code Magician</Title>
49    <PhoneExt>654</PhoneExt>
50    <Email>m.landis@nanonull.com</Email>
51  </Person>

```

Why is this happening:

The value-map component evaluates the input data.

- If the incoming data **matches one** of the entries in the first column, the data is transformed and passed on to the node parameter of substitute-missing, and then on to Title2.
- If the incoming data does not match **any entry** in the left column, then nothing is passed on from value-map to the node parameter i.e. this is an **empty node**.

When this occurs the substitute-missing function retrieves the original node and data from the Title node, and passes it on through the **replace-with** parameter, and then on to Title2.

## 5.10.2 Value-Map component properties

Actions:



Click the insert icon to **insert** a new row before the currently active one.





Click the delete icon to **delete** the currently active row.

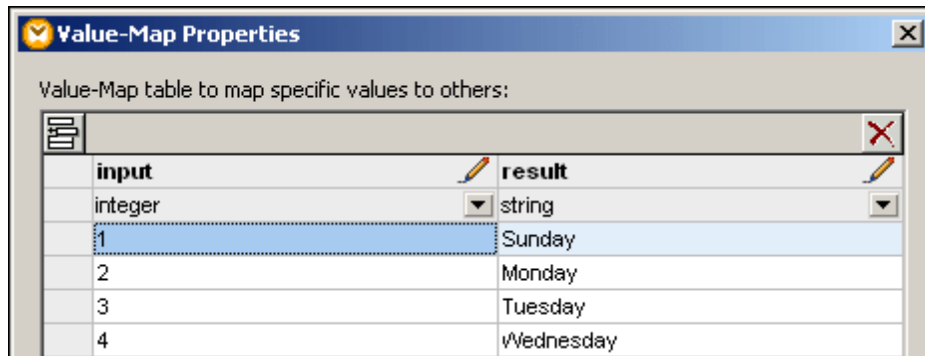


Click the edit icon to **edit** the column header.

You can also reorder lines by dragging them.

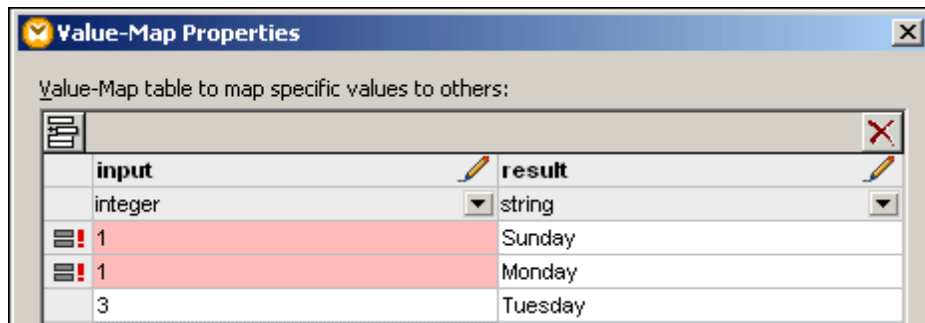
### Changing the column header:

Double clicking the column header, or clicking the pencil icon, allows you to edit the column name and change it to something more meaningful. This will make it easier to identify the purpose of the component, as the column names are also displayed in the mapping.



### Using unique Input values:

The values entered into the input column must be unique. If you enter two identical values, both are automatically highlighted for you to enable you to correct one of them.



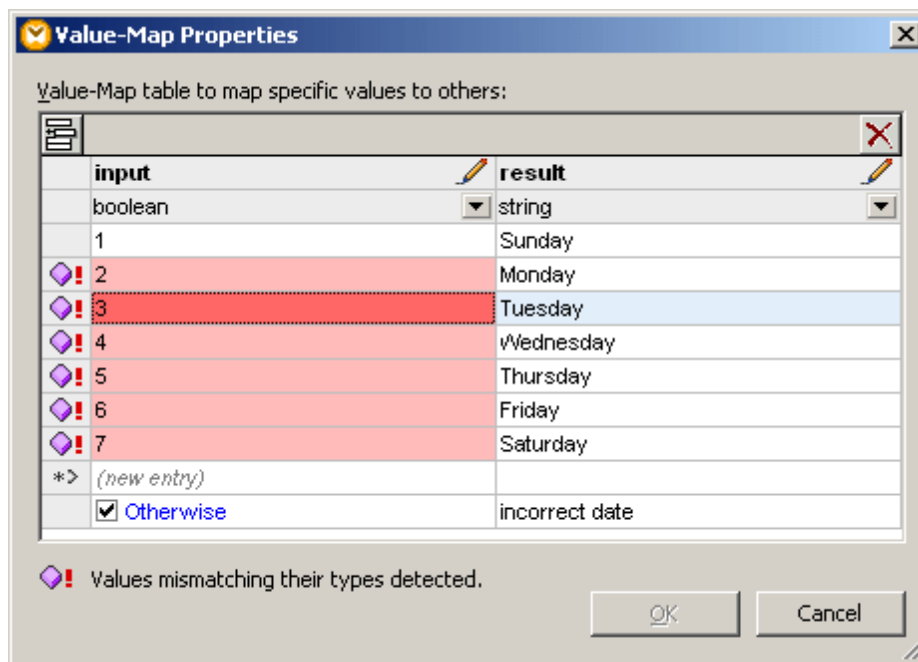
Having corrected one of the values, the OK button is again enabled.

### Input and output datatypes

The input and result datatypes are automatically checked when a selection is made using the combo box. If a mismatch occurs, then the respective fields are highlighted and the OK button is disabled. Change the datatype to one that is supported.

In the screenshot below a boolean and string have been selected.







## 5.11 Mapping Node Names

Most of the time when you create a mapping with MapForce, the goal is to read *values* from a source and write *values* to a target. However, there might be cases when you want to access not only the node *values* from the source, but also the node names. For example, you might want to create a mapping which reads the element or attribute names (not values) from a source XML and converts them to element or attribute values (not names) in a target XML.

Consider the following example: you have an XML file that contains a list of products. Each product has the following format:

```
<product>
  <id>1</id>
  <color>red</color>
  <size>10</size>
</product>
```

Your goal is to convert information about each product into name-value pairs, for example:

```
<product>
  <attribute name="id" value="1" />
  <attribute name="color" value="red" />
  <attribute name="size" value="10" />
</product>
```

For such scenarios, you would need access to the node name from the mapping. With *dynamic* access to node names, which the subject of this topic, you can perform data conversions such as the one above.

**Note:** You can also perform the transformation above by using the [node-name](#) and [static-node-name](#) core library functions. However, in this case, you need to know exactly what element names you expect from the source, and you need to connect every single such element manually to the target. Also, these functions might not be sufficient, for example, when you need to filter or group nodes by name, or when you need to manipulate the data type of the node from the mapping.

Accessing node names dynamically is possible not only when you need to read node names, but also when you need to write them. In a standard mapping, the name of attributes or elements in a target is always known before the mapping runs; it comes from the underlying schema of the component. With dynamic node names, however, you can create new attributes or elements whose name is not known before the mapping runs. Specifically, the name of the attribute or element is supplied by the mapping itself, from any source supported by MapForce.

For dynamic access to a node's children elements or attributes to be possible, the node must actually have children elements or attributes, and it must not be the XML root node.

Dynamic node names are supported when you map to or from the following component types:



- XML
- CSV/FLF\*

\* Requires MapForce Professional or Enterprise Edition.

Dynamic node names are supported in any of the following mapping languages: Built-In\*, XSLT2, XQuery\*, C#\*, C++\*, Java\*.

\* Requires MapForce Professional or Enterprise Edition.

For information about how dynamic node names work, [Getting Access to Node Names](#). For a step-by-step mapping example, see [Example: Map Element Names to Attribute Values](#).

### 5.11.1 Getting Access to Node Names

When a node in an XML component has children nodes, you can get both the name and value of each child node directly on the mapping. This technique is called "dynamic node names". "Dynamic" refers to the fact that processing takes place "on the fly", during mapping runtime, and not based on the static schema information which is known before the mapping runs. This topic provides details on how to enable dynamic access to node names and what you can do with it.

When you read data from a source, "dynamic node names" means that you can do the following:

- Get a list of all children nodes (or attributes) of a node, as a sequence. In MapForce, "sequence" is a list of zero or more items which you can connect to a target and create as many items in the target as there are items in the source. So, for example, if a node has five attributes in the source, you could create five new elements in the target, each corresponding to an attribute.
- Read not only the children node values (as a standard mapping does), but also their names.

When you write data to a target, "dynamic node names" means that you can do the following:

- Create new nodes using names supplied by the mapping (so-called "dynamic" names), as opposed to names supplied by the component settings (so-called "static" names).

To illustrate dynamic node names, this topic makes use of the following XML schema:

**<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\Products.xsd**. This schema is accompanied by a sample instance document, **Products.xml**. To add both the schema and the instance file to the mapping area, select the **Insert | XML Schema/File** menu command and browse for **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\Products.xml**. When prompted to select a root element, choose `products`.

To enable dynamic node names for the `product` node, right-click it and select one of the following context menu commands:

- **Show Attributes with Dynamic Name**, if you want to get access to the node's attributes
- **Show Child Elements with Dynamic Name**, if you want to get access to the node's children elements



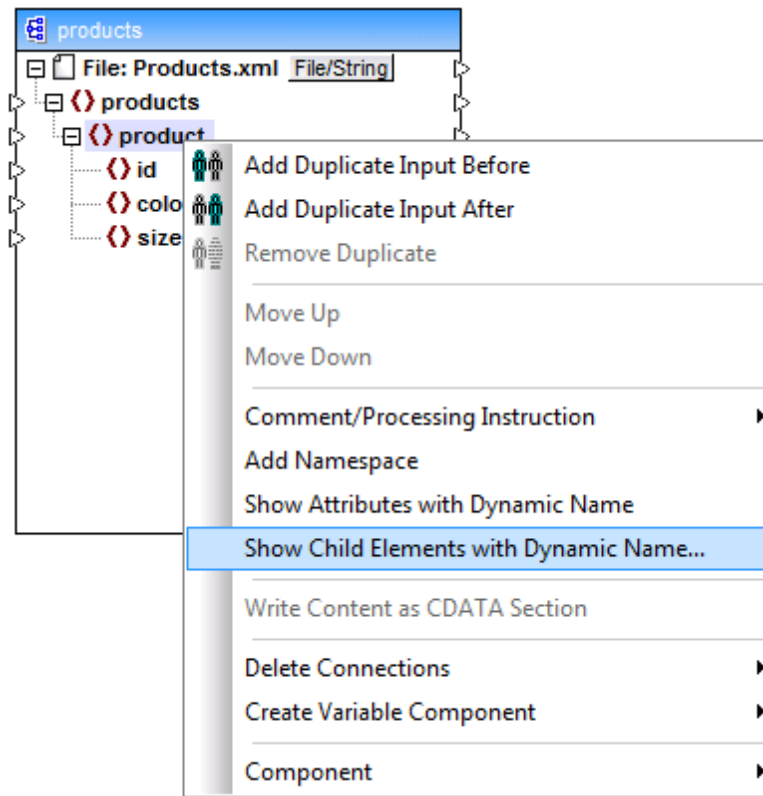


Fig. 1 Enabling dynamic node names (for child elements)

**Note:** The commands above are available only for nodes that have children nodes. Also, the commands are not available for root nodes.

When you switch a node into dynamic mode, a dialog box such as the one below appears. For the purpose of this topic, set the options as shown below; these options are further discussed in [Accessing Nodes of Specific Type](#).



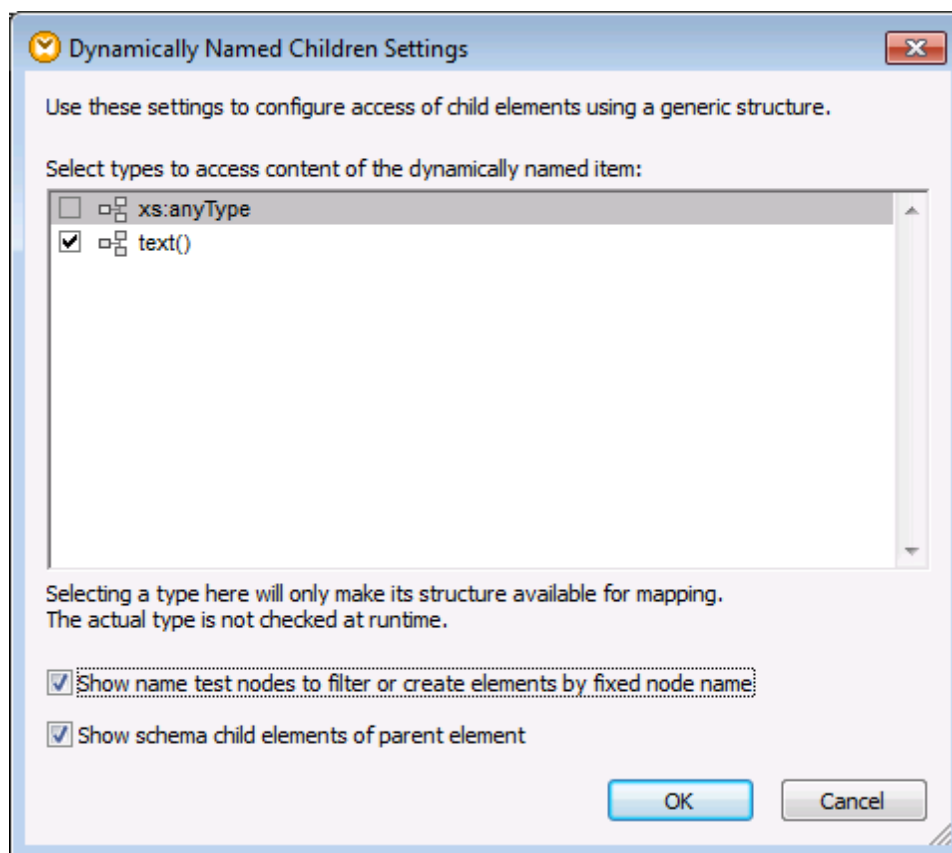


Fig. 2 "Dynamically Named Children Settings" dialog box

Fig. 3 illustrates how the component looks when dynamic node names are enabled for the `product` node. Notice how the appearance of the component has now significantly changed.

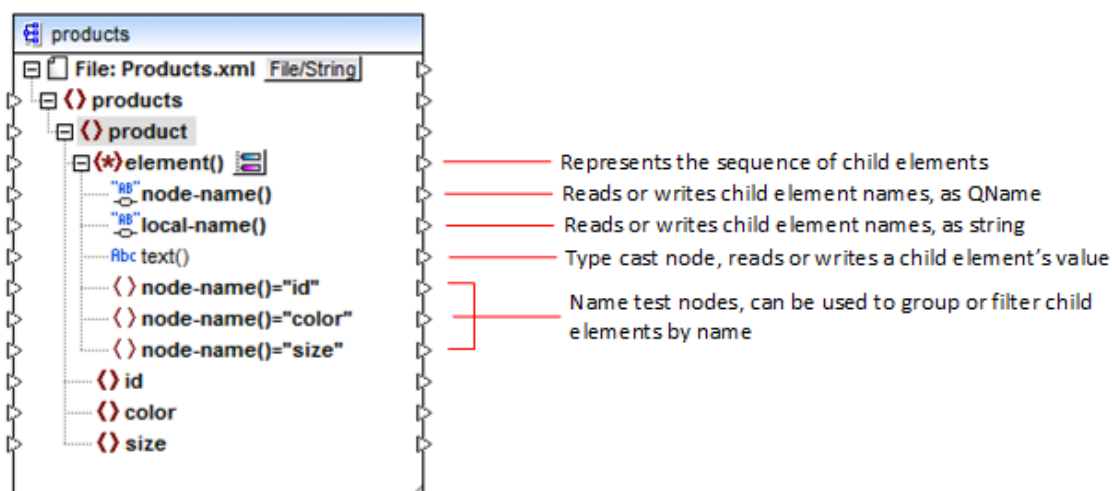


Fig.3 Enabled dynamic node names (for elements)



To switch the component back to standard mode, right-click the `product` node, and disable the option **Show Child Elements with Dynamic Name** from the context menu.

The image below shows how the same component looks when dynamic access to attributes of a node is enabled. The component was obtained by right-clicking the `product` element, and selecting **Show Attributes with Dynamic Name** from the context menu.

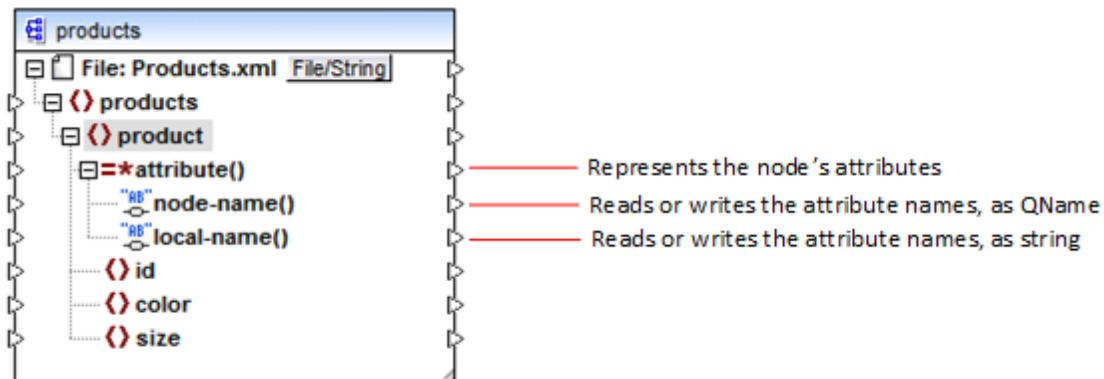


Fig. 4 Enabled dynamic node names (for attributes)

To switch the component back to standard mode, right-click the `product` node, and disable the option **Show Attributes with Dynamic Name** from the context menu.

As illustrated in Fig. 3 and Fig. 4, the component changes appearance when any node (in this case, `product`) is switched into "dynamic node name" mode. The new appearance opens possibilities for the following actions:

- Read or write a list of all children elements or attributes of a node. These are provided by the `element()` or `attribute()` item, respectively.
- Read or write the name of each child element or attribute. The name is provided by the `node-name()` and `local-name()` items.
- In case of elements, read or write the value of each child element, as specific data type. This value is provided by the type cast node (in this case, the `text()` item). Note that only elements can have type cast nodes. Attributes are treated always as "string" type.
- Group or filter child elements by name.

The node types that you can work with in "dynamic node name" mode are described below.

## element()

This node has different behaviour in a source component compared to a target component. In a source component, it supplies the child elements of the node, as a sequence. In Fig.3, `element()` provides a list (sequence) of all children elements of `product`. For example, the sequence created from the following XML would contain three items (since there are three child elements of `product`):

```
<product>
  <id>1</id>
```



```

    <color>red</color>
    <size>10</size>
  </product>

```

Note that the actual name and type of each item in the sequence is provided by the `node-name()` node and the type cast node, respectively (discussed below). To understand this, imagine that you need to transform data from a source XML into a target XML as follows:

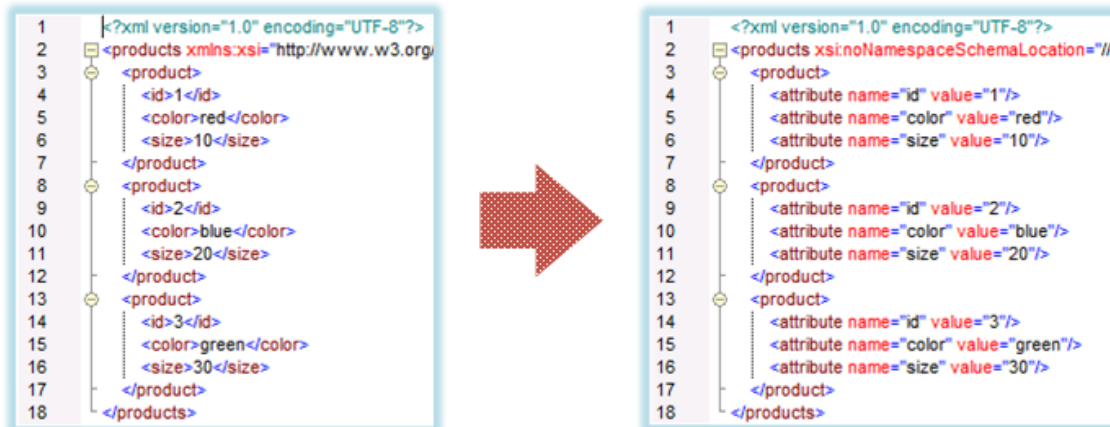


Fig. 6 Mapping XML element names to attribute values (requirement)

The mapping which would achieve this goal looks as follows:

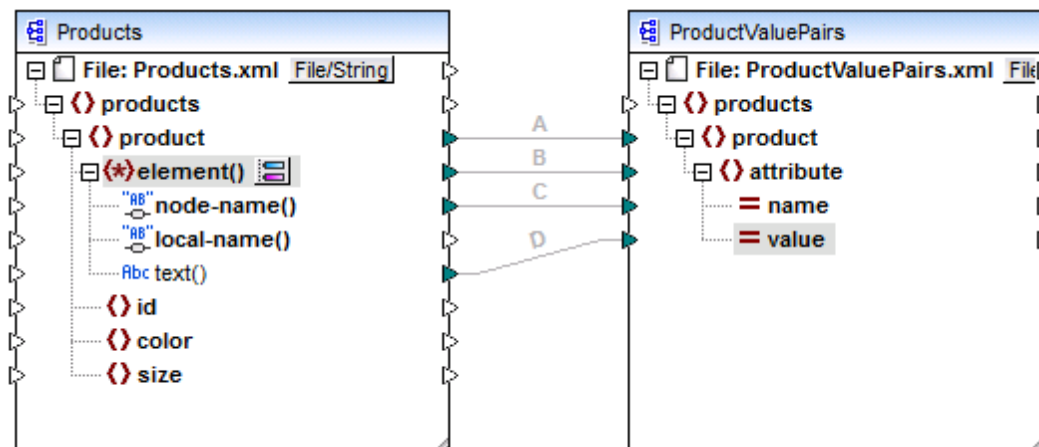


Fig. 7 Mapping XML element names to attribute values (in MapForce)

The role of `element()` here is to supply the sequence of child elements of `product`, while `node-name()` and `text()` supply the actual name and value of each item in the sequence. This mapping is accompanied by a tutorial sample and is discussed in more detail in [Example: Map Element Names to Attribute Values](#).

In a target component, `element()` does not create anything by itself, which is an exception to the basic rule of mapping "for each item in the source, create one target item". The actual elements



are created by the type cast nodes (using the value of `node-name()`) and by name test nodes (using their own name).

### **attribute()**

As shown in Fig. 4, this item enables access to all attributes of the node, at mapping runtime. In a source component, it supplies the attributes of the connected source node, as a sequence. For example, in the following XML, the sequence would contain two items (since `product` has two attributes):

```
<product id="1" color="red" />
```

Note that the `attribute()` node supplies only the value of each attribute in the sequence, always as string type. The name of each attribute is supplied by the `node-name()` node.

In a target component, this node processes a connected sequence and creates an attribute value for each item in the sequence. The attribute name is supplied by the `node-name()`. For example, imagine that you need to transform data from a source XML into a target XML as follows:

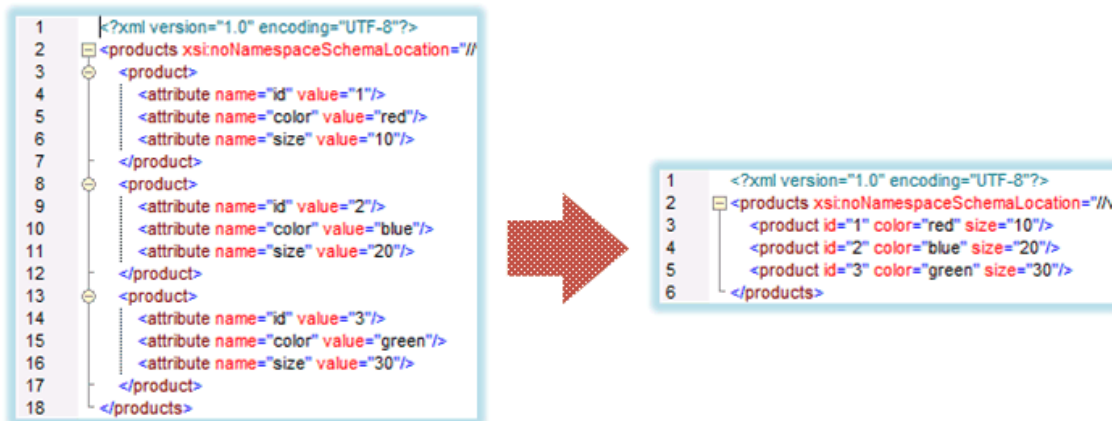


Fig. 8 Mapping attribute values to attribute names (requirement)

The mapping which would achieve this goal looks as follows:



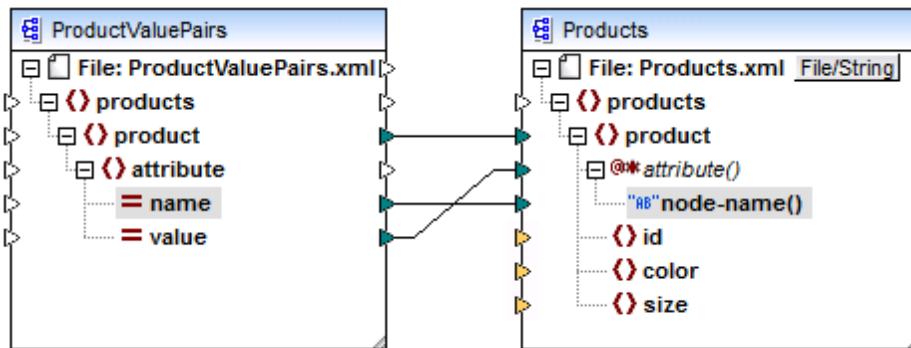


Fig. 9 Mapping attribute values to attribute names (in MapForce)

**Note:** This transformation is also possible without enabling dynamic access to a node's attributes. Here it just illustrates how `attribute()` works in a target component.

If you want to reconstruct this mapping, it uses the same XML components as the **ConvertProducts.mfd** mapping available in the **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\** folder. The only difference is that the target has now become the source, and the source has become the target. As input data for the source component, you will need an XML instance that actually contains attribute values, for example:

```
<?xml version="1.0" encoding="UTF-8"?>
<products>
  <product>
    <attribute name="id" value="1"/>
    <attribute name="color" value="red"/>
    <attribute name="size" value="big"/>
  </product>
</products>
```

Note that, in the code listing above, the namespace and schema declaration have been omitted, for simplicity.

### node-name()

In a source component, `node-name()` supplies the name of each child element of `element()`, or the name of each attribute of `attribute()`, respectively. By default, the supplied name is of type `xs:QName`. To get the name as string, use the `local-name()` node (see Fig. 3).

In a target component, `node-name()` writes the name of each element or attribute contained in `element()` or `attribute()`.


### local-name()


This node works in the same way as `node-name()`, with the difference that the type is `xs:string` instead of `xs:QName`.



### Type cast node

In a source component, the type cast node supplies the value of each child element contained in `element()`. The name and structure of this node depends on the type selected from the "Dynamically Named Children Settings" dialog box (Fig. 2).

To change the type of the node, click the **Change Selection** (  ) button and select a type from the list of available types, including a schema wildcard (`xs:any`). For more information, see [Accessing nodes of specific type](#).


In a target component, the type cast node writes the value of each child element contained in `element()`, as specific data type. Again, the desired data type can be selected by clicking the **Change Selection** (  ) button.

### Name test nodes

In a source component, name test nodes provide a way to group or filter child elements from a source instance by name. You may need to filter child elements by name in order to ensure that the mapping accesses the instance data using the correct type (see [Accessing Nodes of Specific Type](#)).

In general, the name test nodes work almost like normal element nodes for reading and writing values and subtree structures. However, because the mapping semantics is different when dynamic access is enabled, there are some limitations. For example, you cannot concatenate the value of two name test nodes.

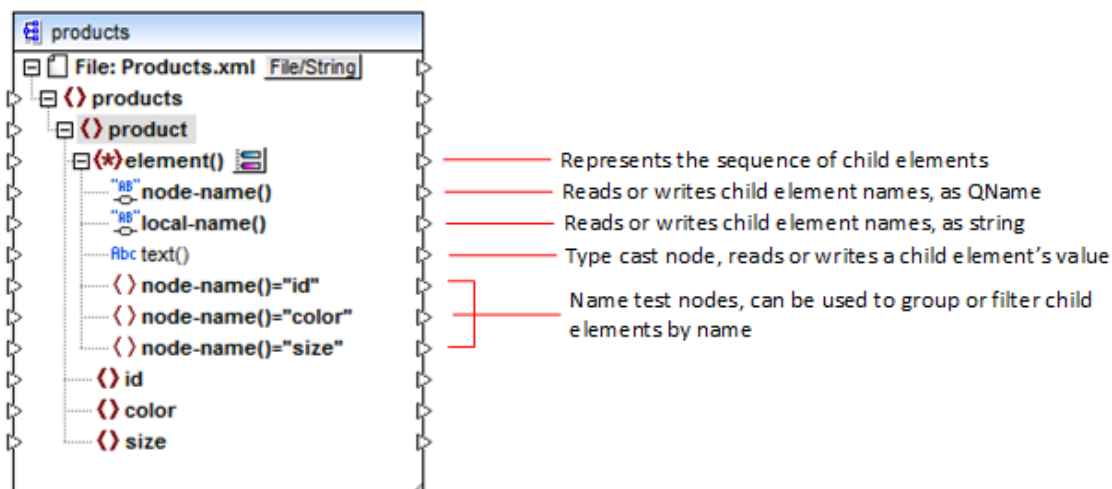
On the target side, name test nodes create as many elements in the output as there are items in the connected source sequence. Their name overrides the value mapped to `node-name()`.


If necessary, you can hide the name test nodes from the component. To do this, click the **Change Selection** (  ) button next to the `element()` node. Then, in the "Dynamically Named Children Settings" dialog box (Fig. 2), clear the **Show name test nodes...** check box.

## 5.11.2 Accessing Nodes of Specific Type

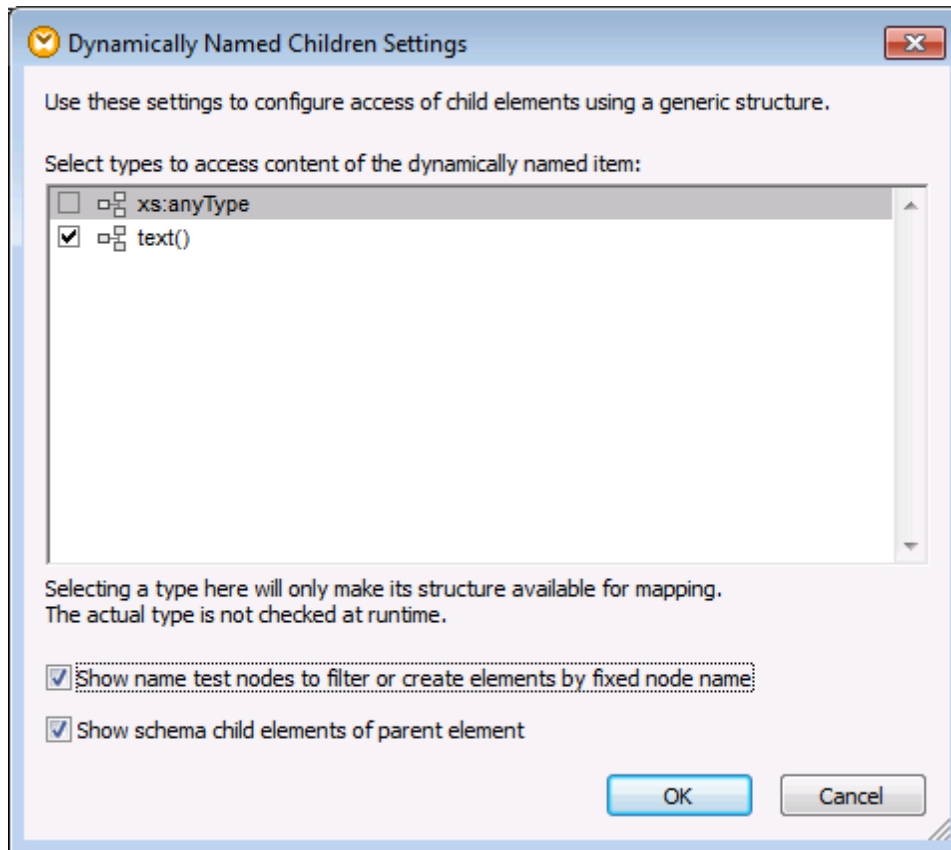
As mentioned in the previous section, [Getting Access to Node Names](#), you can get access to all child elements of a node by right-clicking the node and selecting the **Show Child Elements with Dynamic Name** context menu command. At mapping runtime, this causes the name of each child element to be accessible through the `node-name()` node, while the value—through a special type cast node. In the image below, the type cast node is the `text()` node.





Importantly, the data type of each child element is not known before the mapping runtime. Besides, it may be different for each child element. For example, a `product` node in the XML instance file may have a child element `id` of type `xs:integer` and a child element `size` of type `xs:string`. To let you access the node content of a specific type, the dialog box shown below opens every time when you enable dynamic access to a node's child elements. You can also open this dialog box at any time later, by clicking the **Change Selection** (  ) button next to the `element()` node.





"Dynamically Named Children Settings" dialog box

To access the content of each child element at mapping runtime, you have several options:

1. Access the content as string. To do this, select the **text()** check box on the dialog box above. In this case, a `text()` node is created on the component when you close the dialog box. This option is suitable if the content is of simple type (`xs:int`, `xs:string`, etc.) and is illustrated in the [Example: Map Element Names to Attribute Values](#). Note that a **text()** node is displayed only if a child node of the current node can contain text.
2. Access the content as a particular complex type allowed by the schema. When custom complex types defined globally are allowed by the schema for the selected node, they are also available in the dialog box above, and you can select the check box next to them. In the image above, there are no complex types defined globally by the schema, so none are available for selection.
3. Access the content as any type. This may be useful in advanced mapping scenarios (see "Accessing deeper structures" below). To do this, select the check box next to **xs:anyType**.

Be aware that, at mapping runtime, MapForce (through the type cast node) has no information as to what the actual type of the instance node is. Therefore, your mapping must access the node content using the correct type. For example, if you expect that the node of a source XML instance may have children nodes of various complex types, do the following:

- a) Set the type cast node to be of the complex type that you need to match (see item 2 in



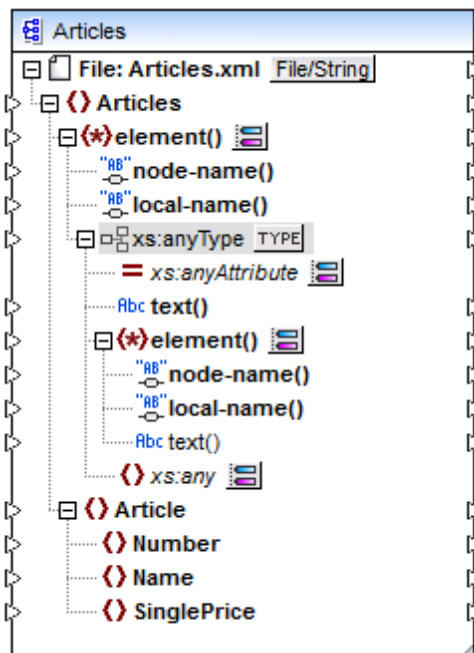
the list above).

b) Add a filter to read from the instance only the complex type that you need to match. For more information about filters, see [Filters and Conditions](#).

## Accessing deeper structures

It is possible to access nodes at deeper levels in the schema than the immediate children of a node. It is useful for advanced mapping scenarios. In simple mappings such as [Example: Map Element Names to Attribute Values](#), you don't need this technique because the mapping accesses only the immediate children of an XML node. However, if you need to access deeper structures dynamically, such as "grandchildren", "grand-grandchildren", and so on, this is possible as shown below.

1. Create a new mapping.
2. On the Insert menu, click **Insert XML Schema/File** and browse for the XML instance file (in this example, the **Articles.xml** file from the **<Documents>\Altova\MapForce2018\MapForceExamplesTutorial\** folder).
3. Right-click the **Articles** node and select the **Show Child Elements with Dynamic Name** context command.
4. Select **xs:anyType** from the "Dynamically Named Children Settings" dialog box.
5. Right-click the **xs:anyType** node and select again the **Show Child Elements with Dynamic Name** context command.
6. Select **text()** from the "Dynamically Named Children Settings" dialog box.



In the component above, notice there are two `element()` nodes. The second `element()` node provides dynamic access to grandchildren of the `<Articles>` node in the **Articles.xml** instance.

```
<?xml version="1.0" encoding="UTF-8"?>
```



```

<Articles xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Articles.xsd">
  <Article>
    <Number>1</Number>
    <Name>T-Shirt</Name>
    <SinglePrice>25</SinglePrice>
  </Article>
  <Article>
    <Number>2</Number>
    <Name>Socks</Name>
    <SinglePrice>2.30</SinglePrice>
  </Article>
  <Article>
    <Number>3</Number>
    <Name>Pants</Name>
    <SinglePrice>34</SinglePrice>
  </Article>
  <Article>
    <Number>4</Number>
    <Name>Jacket</Name>
    <SinglePrice>57.50</SinglePrice>
  </Article>
</Articles>




```

*Articles.xml*

For example, to get "grandchildren" element names (Number, Name, SinglePrice), you would draw a connection from the `local-name()` node under the `second element()` node to a target item. Likewise, to get "grandchildren" element values (1, T-Shirt, 25), you would draw a connection from the `text()` node.

Although not applicable to this example, in real-life situations, you can further enable dynamic node names for any subsequent `xs:anyType` node, so as to reach even deeper levels.

Note the following:

- The **TYPE** button allows you to select any derived type from the current schema and display it in a separate node. This may only be useful if you need to map to or from derived schema types (see [Derived XML Schema Types](#)).
- The **Change Selection** (  ) button next to an `element()` node opens the "Dynamically Named Children Settings" dialog box discussed in this topic.
- The **Change Selection** (  ) button next to `xs:anyAttribute` allows you to select any attribute defined globally in the schema. Likewise, the **Change Selection** (  ) button next to `xs:any` element allows you to select any element defined globally in the schema. This works in the same way as mapping to or from schema wildcards (see also [Wildcards - xs:any / xs:anyAttribute](#)). If using this option, make sure that the selected attribute or element can actually exist at that particular level according to the schema.



### 5.11.3 Example: Map Element Names to Attribute Values

This example shows you how to map element names from an XML document to attribute values in a target XML document. The example is accompanied by a sample mapping, which is available at the following path: **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\ConvertProducts.mfd**.

To understand what the example does, let's assume you have an XML file that contains a list of products. Each product has the following format:

```
<product>
  <id>1</id>
  <color>red</color>
  <size>10</size>
</product>
```

Your goal is to convert information about each product into name-value pairs, for example:

```
<product>
  <attribute name="id" value="1" />
  <attribute name="color" value="red" />
  <attribute name="size" value="10" />
</product>
```

To perform a data mapping such as the one above with minimum effort, this example uses a MapForce feature known as "dynamic access to node names". "Dynamic" means that, when the mapping runs, it can read the node names (not just values) and use these names as values. You can create the required mapping in a few simple steps, as shown below.

#### Step 1: Add the source XML component to the mapping

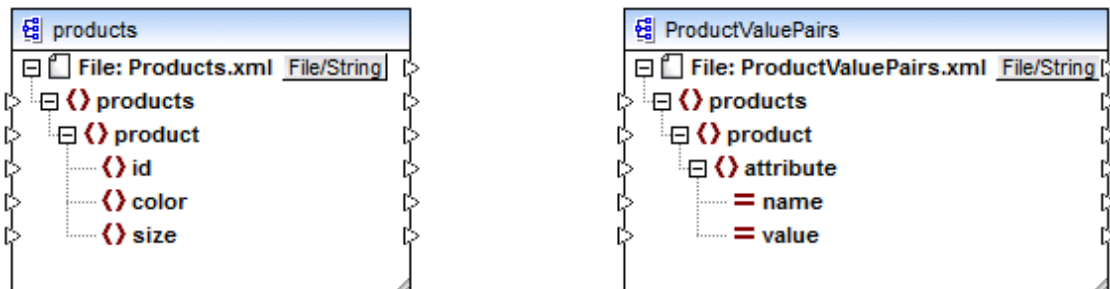
- On the **Insert** menu, click **XML Schema/File**, and browse for the following file:  
**<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\Products.xml**.  
This XML file points to the **Products.xsd** schema located in the same folder.

#### Step 2: Add the target XML component to the mapping

- On the **Insert** menu, click **XML Schema/File**, and browse for the following schema file:  
**<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\ProductValuePairs.xsd**. When prompted to supply an instance file, click **Skip**. When prompted to select a root element, select **products** as root element.

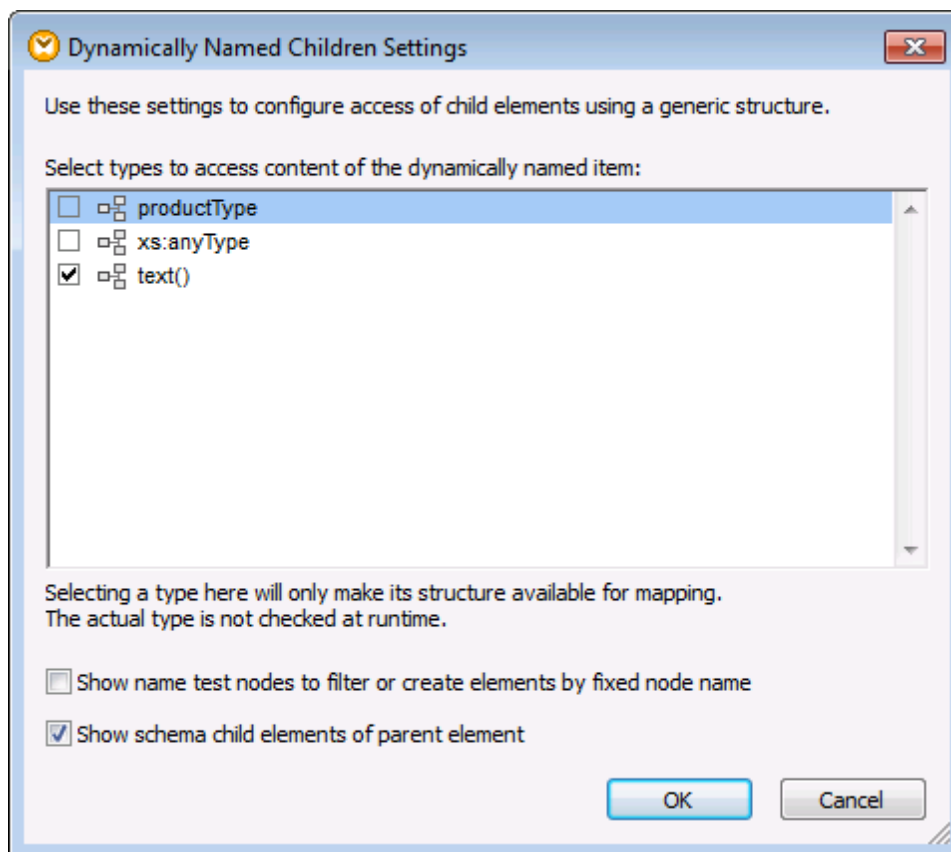
At this stage, the mapping should look as follows:





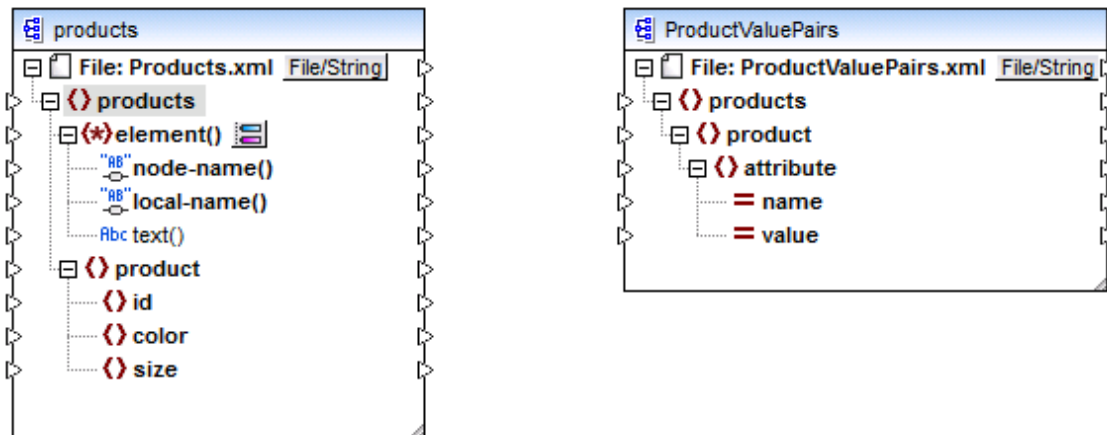
### Step 3: Enable dynamic access to child nodes

1. Right-click the `products` node on the source component, and select **Show Child Elements with Dynamic Name** from the context menu.
2. In the dialog box which opens, select **text()** as type. Leave other options as is.



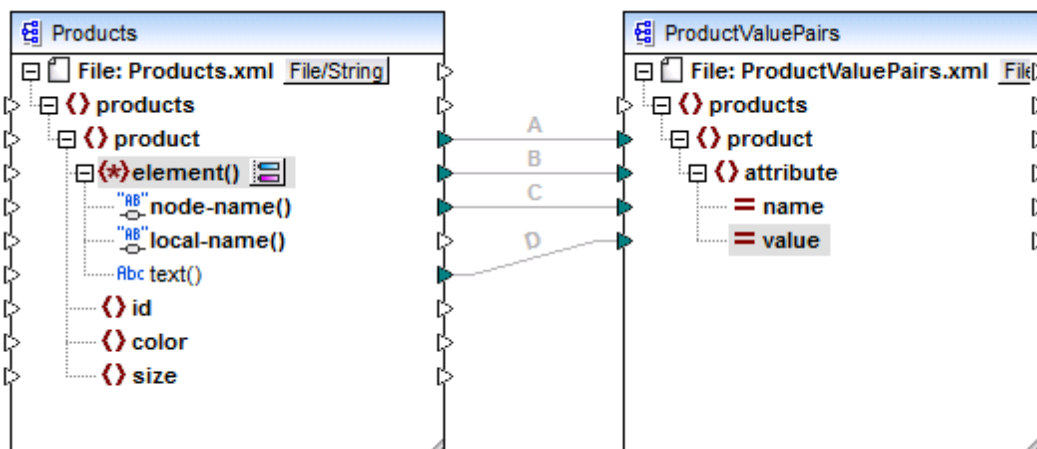
Notice that a `text()` node has been added on the source component. This node will supply the content of each child item to the mapping (in this case, the value of "id", "color", and "size").





#### Step 4: Draw the mapping connections

Finally, draw the mapping connections A, B, C, D as illustrated below. Optionally, double-click each connection, starting from the top one, and enter the text "A", "B", "C", and "D", respectively, into the Description box.



*ConvertProducts.mfd*

In the mapping illustrated above, connection A creates, for each product in the source, a product in the target. So far, this is a standard MapForce connection that does not address the node names in any way. The connection B, however, creates, for each encountered child element of `product`, a new element in the target called `attribute`.

Connection B is a crucial connection in the mapping. To reiterate the goal of this connection, it carries a *sequence* of child elements of `product` from the source to the target. It does not carry the actual *names* or *values*. Therefore, it must be understood as follows: if the source **element()** has N child elements, create N instances of that item in the target. In this



particular case, `product` in the source has three children elements (`id`, `color` and `size`). This means that each `product` in the target will have three child elements with the name `attribute`.

Although not illustrated in this example, the same rule is used to map child elements of **attribute()**: if the source **attribute()** item has N child attributes, create N instances of that item in the target.

Next, connection C copies the actual name of each child element of `product` to the target (literally, "id", "color", and "size").

Finally, connection D copies the value of each child element of `product`, as string type, to the target.

To preview the mapping output, click the **Output** tab and observe the generated XML. As expected, the output contains several products whose data is stored as name-value pairs, which was the intended goal of this mapping.

```
<?xml version="1.0" encoding="UTF-8"?>
<products xsi:noNamespaceSchemaLocation="ProductValuePairs.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <product>
    <attribute name="id" value="1"/>
    <attribute name="color" value="red"/>
    <attribute name="size" value="10"/>
  </product>
  <product>
    <attribute name="id" value="2"/>
    <attribute name="color" value="blue"/>
    <attribute name="size" value="20"/>
  </product>
  <product>
    <attribute name="id" value="3"/>
    <attribute name="color" value="green"/>
    <attribute name="size" value="30"/>
  </product>
</products>
```

*Generated mapping output*



## 5.12 Mapping Rules and Strategies

MapForce generally maps data in an intuitive way, but you may come across situations where the resulting output seems to have too many, or too few items. This topic is intended to help you avoid such mapping problems.

### General rule

Generally, every connection between a source and target item means: for each source item, create one target item. If the source node contains simple content (for example, string or integer) and the target node accepts simple content, then MapForce copies the content to the target node and, if necessary, converts the data type.

This generally holds true for all connections, with the following exceptions:

- A target XML root element is always created once and only once. If you connect a sequence to it, only the contents of the element will be repeated, but not the root element itself, and the result might not be schema-valid. If attributes of the root element are also connected, the XML serialization will fail at runtime, so you should avoid connecting a sequence to the root element. If what you want to achieve is creating multiple output files, connect the sequence to the "File" node instead, via some function that generates file names.
- Some nodes accept a single value, not a sequence (for example, XML attributes, , and output components in user-defined functions).

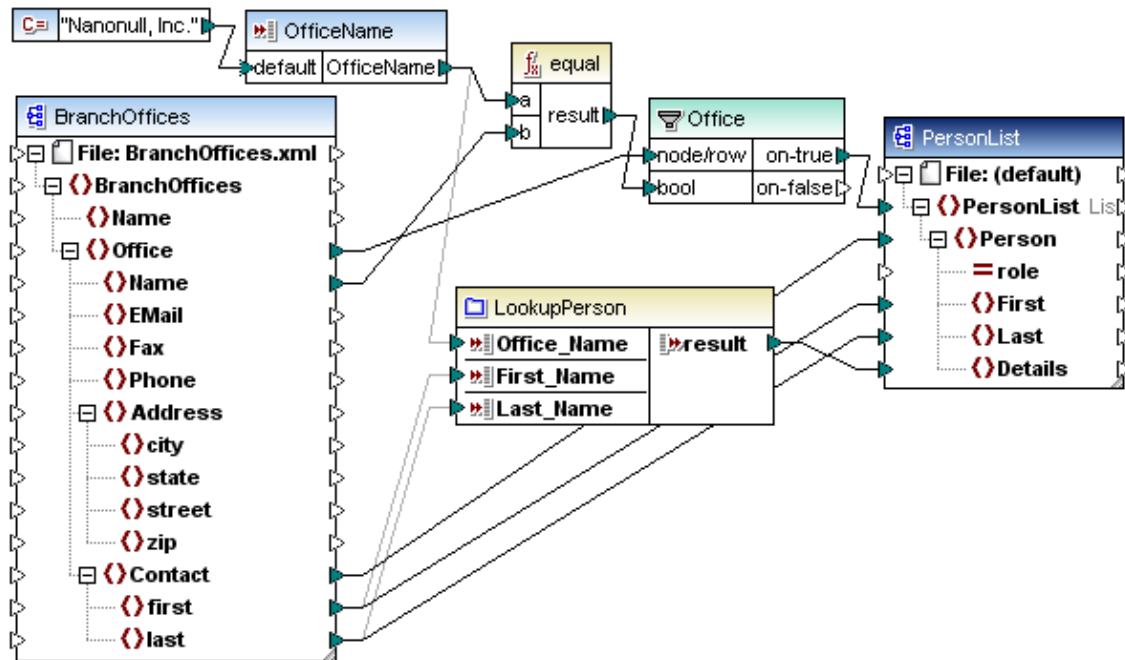
### The "context" and "current" items

MapForce displays the structure of a schema file as a hierarchy of mappable items in the component. Each of these nodes may have many instances (or none) in the instance file or database.

Example: If you look at the source component in **PersonListByBranchOffice.mfd**, there is only a single node **first** (under **Contact**). In the **BranchOffices.xml** instance file, there are multiple **first** nodes and **Contact** nodes having different content, under different **Office** parent nodes.

It depends on the current **context** (of the **target** node) which source nodes are actually selected and have their data copied, via the connector, to the target component/item.





*PersonListByBranchOffice.mfd*

This context is defined by the **current target node** and the connections to its ancestors:

- Initially the context contains only the source components, but no specific nodes. When evaluating the mapping, MapForce processes the **target root** node first (PersonList), then works down the hierarchy.
- The connector to the **target** node is traced back to all source items directly or indirectly connected to it, even via functions that might exist between the two components. The source items and functions results are added to the context for this node.
- For each new target node a new context is established, that initially contains all items of the parent node's context. Target sibling nodes are thus independent of each other, but have access to all source data of their parent nodes.

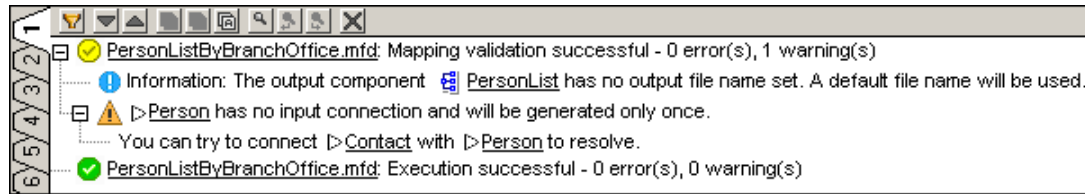
Applied to the example mapping above (**PersonListByBranchOffice.mfd**):

- The connection from **Office** through the filter (Office) to **PersonList** defines a **single** office as the context for the whole target document (because PersonList is the root element of the target component). The office name is supplied by the input component, which has a default containing "Nanonull, Inc."
- All connections/data to the **descendants** of the root element PersonList, are automatically affected by the filter condition, because the selected single office is in the context.
- The connection from **Contact** to **Person** creates one target Person **per** Contact item of the source XML (general rule). For each Person one specific Contact is added to the context, from which the children of Person will be created.
- The connector from **first** to **First** selects the first name of the current Contact and writes it to the target item First.

Leaving out the connector from **Contact** to **Person** would create only **one** Person with multiple



First, Last, and Detail nodes, which is not what we want here. In such situations, MapForce issues a warning and a suggestion to fix the problem: "You can try to connect Contact with Person to resolve":



## Sequences

MapForce displays the structure of a schema file as a hierarchy of mappable items in the component.

Depending on the (target) context, each mappable item of a source component can represent:

- a **single instance** node of the assigned input file
- a **sequence** of zero to **multiple instance** nodes of the input file

If a sequence is connected to a **target** node, a loop is created to create as many target nodes as there are source nodes.

If a **filter** is placed between the sequence and target node, the bool condition is checked for each input node i.e. each item in the sequence. More exactly, a check is made to see if there is at least one bool in each sequence that evaluates to true. The priority context setting can influence the order of evaluation, see below.

As noted above, filter conditions automatically apply to all descendant nodes.

**Note:** If the source schema specifies that a specific node occurs exactly once, MapForce may remove the loop and take the first item only, which it knows must exist. This optimization can be disabled in the source Component Settings dialog box (check box "Enable input processing optimizations based on min/maxOccurs").

**Function inputs** (of normal, non-sequence functions) work similar to target nodes: If a sequence is connected to such an input, a loop is created around the function call, so it will produce as **many results** as there are items in the sequence.

If a sequence is connected to **more than one** such function input, MapForce creates nested loops which will process the **Cartesian product** of all inputs. Usually this is not desired, so only one single sequence with multiple items should be connected to a function (and all other parameters bound to singular current items from parents or other components).

**Note:** If an empty sequence is connected to such a function (e.g. concat), you will get an **empty sequence** as result, which will produce no output nodes at all. If there is no result in your target output because there is no input data, you can use the "substitute-missing" function to insert a substitute value.

Functions with **sequence inputs** are the only functions that can produce a result if the input sequence is **empty**:

- **exists**, **not-exists** and **substitute-missing** (also, **is-not-null**, **is-null** and **substitute-null**, which are aliases for the first three)



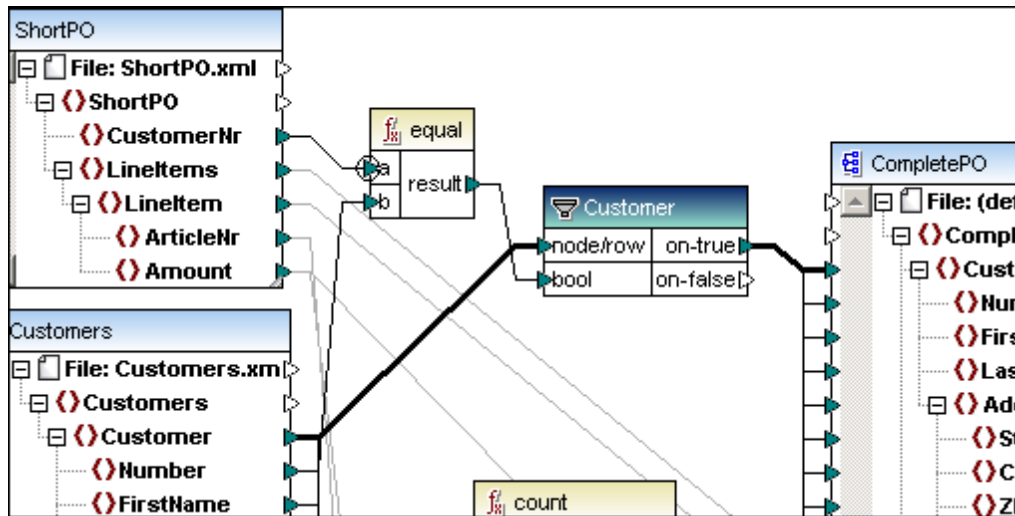
- aggregate functions (**sum**, **count**, etc.)
- regular user-defined functions that accept sequences (i.e. non-inlined functions)

The sequence input to such functions is always evaluated independently of the current target node in the context of its ancestors. This also means that any filter components connected to such functions, do not affect any other connections.

### Priority context

Usually, function parameters are evaluated from top to bottom, but it is possible to define one parameter to be evaluated before all others, using the **priority context** setting.

In functions connected to the bool input of **filter** conditions, the priority context affects not only the comparison function itself but also the evaluation of the filter, so it is possible to join together two source sequences (see CompletePO.mfd, CustomerNo and Number).

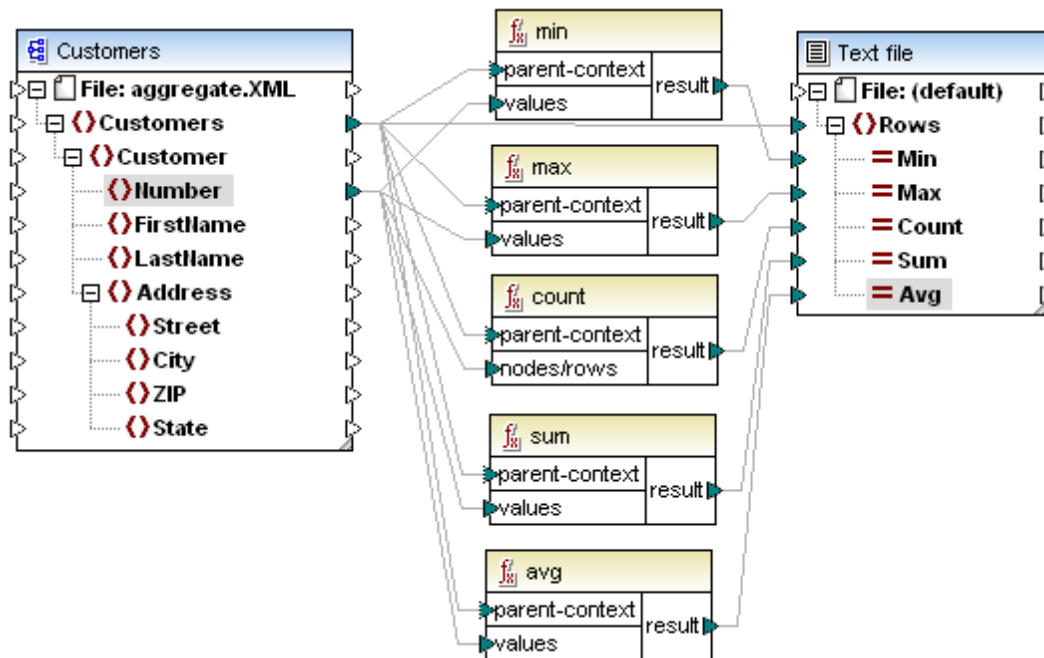


In this example, the priority context forces ShortPO/CustomerNr to be evaluated before iterating and filtering the Customer nodes from the Customers component. See [Priority Context node/item](#)

### Overriding the context

Some [aggregate functions](#) have an optional “parent-context” input. If this input is not connected, it has no effect and the function is evaluated in the normal context for sequence inputs (that is, in the context of the target node's parent).





If the `parent-context` input is connected to a source node, the function is evaluated for each `parent-context` node and will produce a separate result for each occurrence. See also [Overriding the Mapping Context](#).

### Bringing multiple nodes of the same source component into the context

This is required in some special cases and can be done with [Intermediate variables](#).

## 5.12.1 Changing the Processing Order of Mapping Components

MapForce supports mappings that have several target components. Each of the target components has a preview button allowing you to preview the mapping result for that specific component.

If the mapping is executed from the command line or from generated code, then, regardless of the currently active preview, the full mapping is executed and the output for each target component is generated.

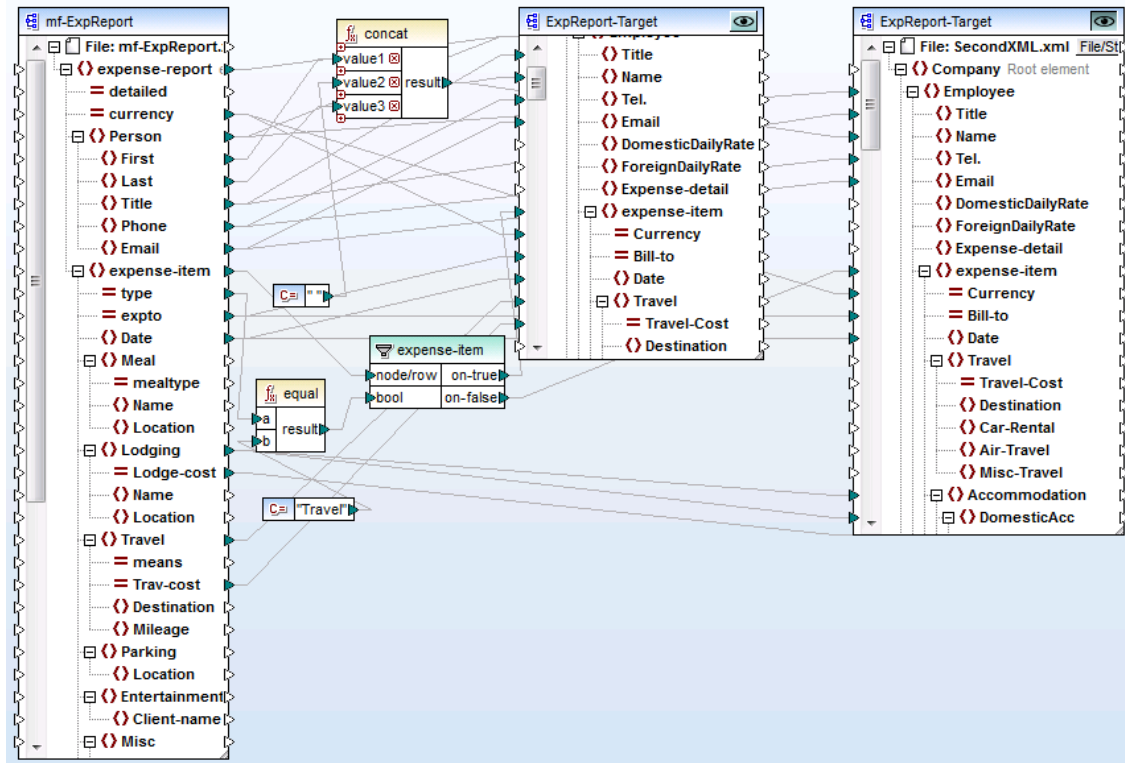
The order in which the target components are processed can be directly influenced by changing the position of target components in the mapping window. The **position** of a component is defined as its top left corner.

Target components are processed according to their Y-X position on screen, from top to bottom and left to right.

- If two components have the same vertical position, then the leftmost takes precedence.
- If two component have the same horizontal position, then the highest takes precedence.
- In the unlikely event that components have the exact same position, then an unique internal component ID is automatically used, which guarantees a well-defined order but which cannot be changed.



The screenshot below shows the tutorial sample **Tut-ExpReport-multi.mfd** available in the **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\** folder. Both target components (ExpReport-Target) have the same **vertical** position, and the preview button is active on the right hand target component.



*Tut-ExpReport-multi.mfd (MapForce Enterprise Edition)*

Having selected XSLT2 and generated the code:

- The leftmost target component is processed first and generates the **ExpReport.xml** file.
- The component to the right of it is processed next and generates the **SecondXML.xml** file.

You can check that this is the case by opening the **DoTransform.bat** file (in the output folder you specified) and see the sequence the output files are generated. **ExpReport-Target.xml** is the first output to be generated by the batch file, and **SecondXML.xml** the second.

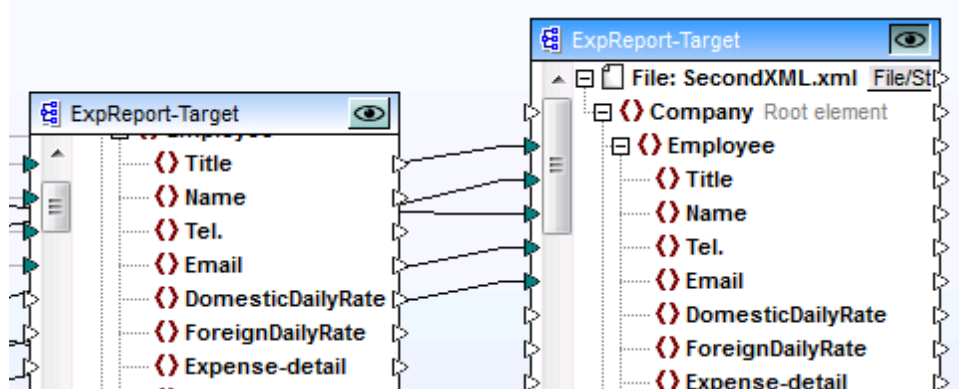
```
@echo off

RaptorXML xslt --xslt-version=2 --
input="C:\Users\me\Documents\Altova\MapForce2013\MapForceExamples\Tutorial\m
f-ExpReport.xml" --
output="C:\Users\me\Documents\Altova\MapForce2013\MapForceExamples\Tutorial\
ExpReport-Target.xml" %* "MappingMapToExpReport-Target.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --
input="C:\Users\me\Documents\Altova\MapForce2013\MapForceExamples\Tutorial\m
f-ExpReport.xml" --
output="C:\Users\me\Documents\Altova\MapForce2013\MapForceExamples\Tutorial\
SecondXML.xml" %* "MappingMapToExpReport-Target2.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
```



### Changing the mapping processing sequence:

1. Click the left target component and move it below the one at right.



2. Regenerate your code and take a look at the **DoTransform.bat** file.

```
@echo off
RaptorXML xslt --xslt-version=2 --
input="C:\Users\alp\Documents\Altova\MapForce2013\MapForceExamples\Tutorial\
mf-ExpReport.xml" --
output="C:\Users\alp\Documents\Altova\MapForce2013\MapForceExamples\Tutorial\
SecondXML.xml" %* "MappingMapToExpReport-Target.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --
input="C:\Users\alp\Documents\Altova\MapForce2013\MapForceExamples\Tutorial\
mf-ExpReport.xml" --
output="C:\Users\alp\Documents\Altova\MapForce2013\MapForceExamples\Tutorial\
ExpReport-Target.xml" %* "MappingMapToExpReport-Target2.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
```

**SecondXML.xml** is now the first output to be generated by the batch file, and **ExpReport-Target.xml** the second.

### Chained mappings

The same processing sequence as described above is followed for [chained mappings](#). The chained mapping group is taken as one unit however. Repositioning the intermediate or final target component of a single chained mapping has no effect on the processing sequence.

Only if multiple "chains" or multiple target components exist in a mapping does the position of the **final** target components of each group determine which is processed first.

- If two final target components have the same vertical position, then the leftmost takes precedence.
- If two final target component have the same horizontal position, then the highest takes precedence.
- In the unlikely event that components have the exact same position, then an unique internal component ID is automatically used, which guarantees a well-defined order but which cannot be changed.



### 5.12.2 Priority Context node/item

When applying a function to different items in a schema, MapForce needs to know what the context node will be. All other items are then processed relative to this one. This is achieved by designating the item (or node) as the priority context.

Priority-context is used to prioritize execution when mapping unrelated items.

Mappings are always executed top-down; if you loop/search through two sources then each loop is processed consecutively. When mapping unrelated elements, without setting the priority context, MapForce does not know which loop needs to be executed first, it therefore automatically selects the first source.

Solution:

Decide which source data is to be looped/searched first, and then set the priority context on the connector to that source data.

The **CompletePO.mfd** file available in the [...\MapForceExamples](#) folder, is shown below.

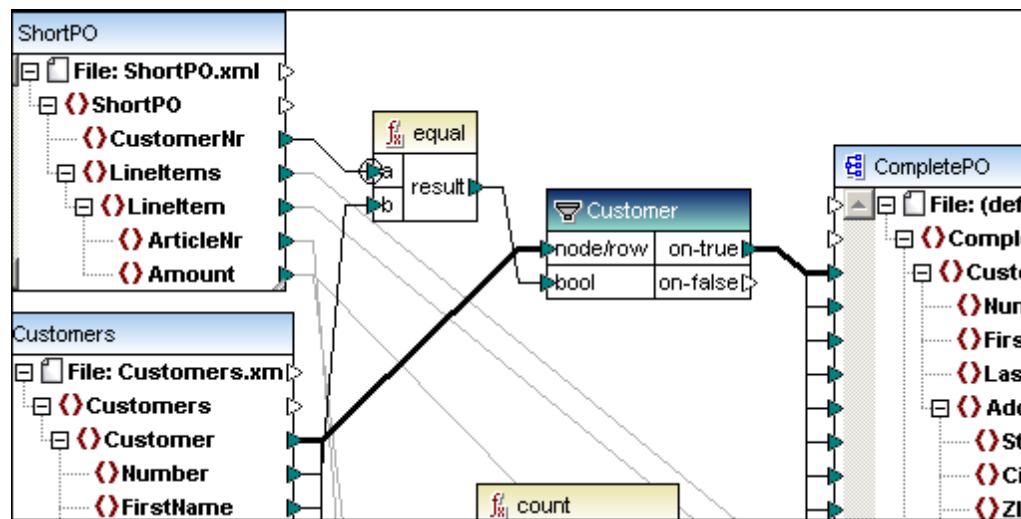
Please note that there are multiple source components in this example. **ShortPO, Customers, and Articles** are all schemas with associated XML instance files. The data from each, are then mapped to the CompletePO schema / XML file. The priority context icon, is enclosed in a circle as a visual indication.

- The **CustomerNr** in ShortPO is compared with the item **Number** in the Customers file.
- **CustomerNr** has been designated as the **priority context**, and is placed in the **a** parameter of the equal function.
- The **Customers** file is then searched (**once**) for the **same** number. The **b** parameter contains the Number item from the Customers file.
- If the number is found, then the result is passed to the **bool** parameter of the **filter** function.
- The **node/row** parameter passes on the **Customers** data to "on-true" when the bool parameter is true, i.e. when the same number has been found.
- The rest of the customer data is then passed on as: Number, FirstName, LastName items, are all connected to the corresponding items in the target schema.

Designating the **b** parameter of the equal function (i.e. item Number), as the **priority context** would cause:

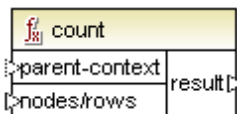
- MapForce to load the first Number into the **b** parameter
- Check against the **CustomerNr** in **a**, if not equal,
- Load the next Number into **b**, check against **a**, and
- Iterate through every Number in the file while trying to find that number in ShortPO.





### 5.12.3 Overriding the Mapping Context

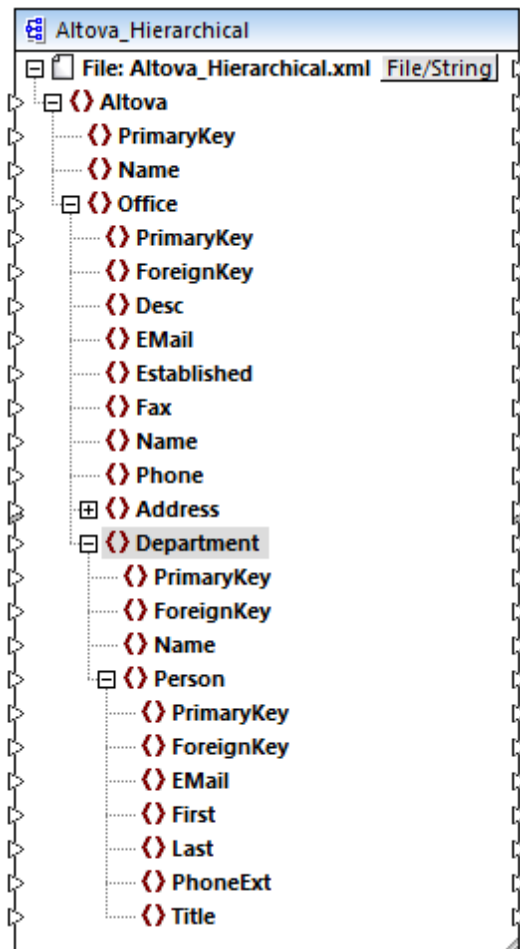
In some mappings, in order to achieve the desired mapping output, it may be necessary to override the mapping context. For this reason, some components provide an optional `parent-context` item in their structure which enables you to influence the mapping context if so required. Examples of such components are aggregate functions and variables.



*An aggregate function with optional parent-context*

To understand why the mapping context is important, let's add to the mapping an XML file that contains nested nodes with multiple levels. On the **Insert** menu, click **XML Schema/File**, and browse for the file: **<Documents>\Altova\MapForce2018\MapForceExamples\Altova\_Hierarchical.xml**.





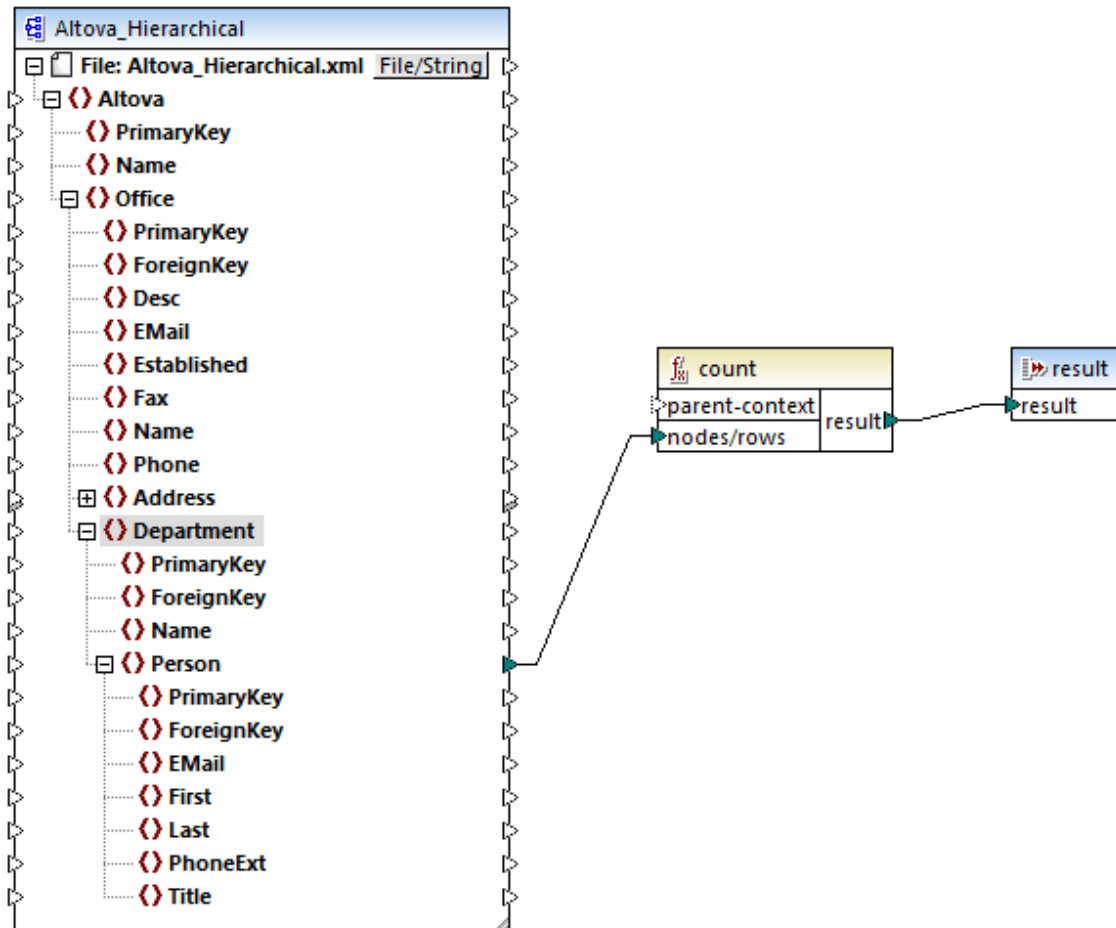
Altova\_Hierarchical.xml

Importantly, in the XML file above, the `Office` parent node contains multiple `Department` nodes, and each `Department` contains multiple `Person` nodes. If you open the actual XML file in an XML editor, you can see that the distribution of people by office and department is as follows:

Office	Department	Number of people
Nanonull, Inc.	Administration	3
	Marketing	2
	Engineering	6
	IT & Technical Support	4
Nanonull Partners, Inc.	Administration	2
	Marketing	1
	IT & Technical Support	3



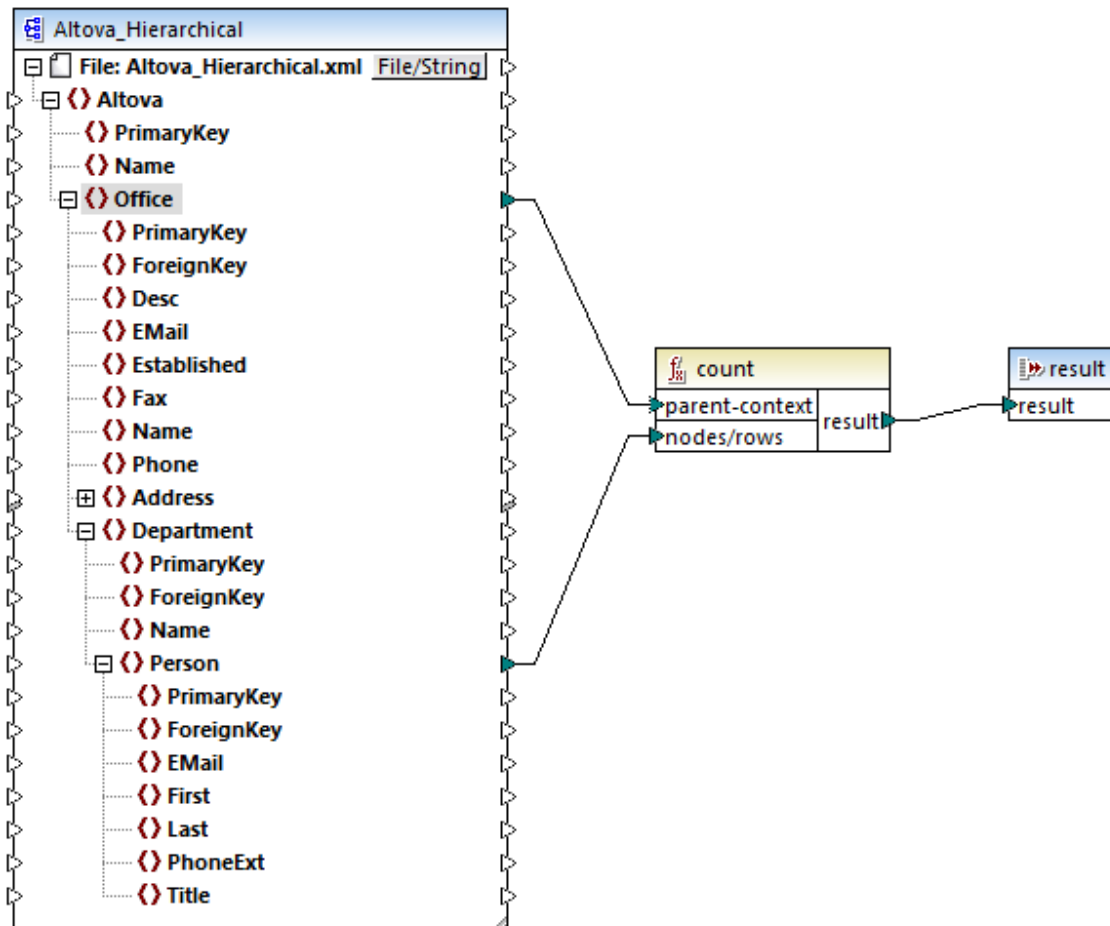
Now let's assume that your mapping should count all people in all departments. To achieve this requirement, you can add the **count** function from [core | aggregate functions](#) and map data as follows:



If you preview the mapping at this stage, the output is 21, which corresponds to the total number of people in all departments. Notice that the **count** function includes an optional **parent-context** item, which so far has not been connected. As a result, the parent context of the **count** function is the default root node of the source component (which, in this case, is the **Altova** item). This means that all the persons, from all departments, are considered for the scope of the **count** function. This is the way the mapping context works by default, as outlined in [Mapping Rules and Strategies](#), and this is sufficient in most mapping scenarios.

However, it is possible to override the default mapping context if necessary. To do this, add a connection from the **Department** node to the **parent-context** item as shown below.



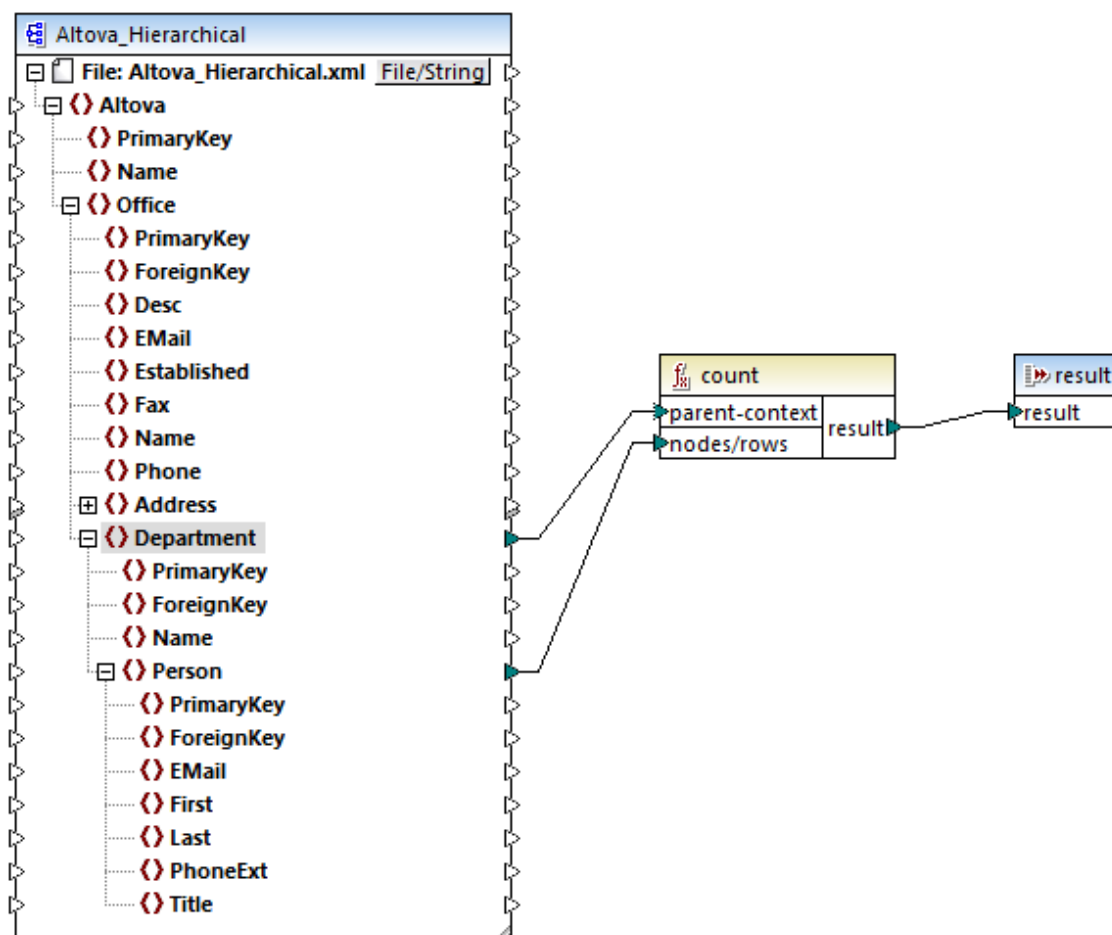


By changing the mapping as shown above, you are instructing the mapping to iterate over people records *in the context of each office*. Therefore, if you preview the mapping now, the output will be 15\*. This is exactly the number of people in the first office, "Nanonull, Inc.". The explanation is that this time the people nodes were counted twice (once for each office). The count of people in each office was 15 and 6, respectively. However, only the first result was returned (because the function cannot return a sequence of values, only a simple value).

\* Assuming that the target language of the mapping is other than XSLT 1.0.

You can further modify the mapping so as to change the mapping context to `Department`, as shown below. This time the people records would be counted in the context of each department (that is, 7 times, which corresponds to the total number of departments). Again, only the first of the results is returned, so the mapping output is 3, which corresponds to the number of people in the first department of the first office.





While this mapping is not doing much yet, its point is to illustrate how the `parent-context` item influences the output of the mapping. Having this in mind, you can override the `parent-context` in other mappings, such as those that contain variables. See also [Example: Grouping and Subgrouping Records](#).







# Chapter 6

---

## Data Sources and Targets



## 6 Data Sources and Targets

This section provides information specific to various source and target component types that MapForce can map from or to:

- [XML and XML Schema](#)
- [HL7 Version 3](#)



## 6.1 XML and XML schema

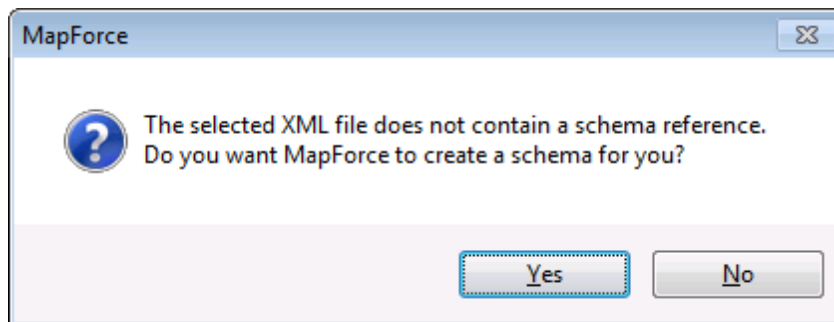
Altova website: [XML mapping](#)

In the introductory part of this documentation, you have seen examples of simple mappings that use XML and XML schema files as source or target components. This section provides further information about using XML components in your mappings. It includes the following topics:

- [XML Component Settings](#)
- [Using DTDs as "schema" components](#)
- [Derived XML Schema types - mapping to](#)
- [QName support](#)
- [Nil Values / Nillable](#)
- [Comments and Processing Instructions](#)
- [CDATA sections](#)
- [Wildcards - xs:any](#)

### 6.1.1 Generating an XML Schema

MapForce can automatically generate an XML schema based on an existing XML file if the XML Schema is not available. Whenever you add to the mapping area an XML file without a schema (using the menu command **Insert | XML Schema/File**), the following dialog box appears.



Click **Yes** to generate the schema, you will then be prompted to select the directory where the generated schema should be saved.

When MapForce generates a schema from an XML file, data types for elements/attributes must be inferred from the XML instance document and may not be exactly what you expect. It is recommended that you check whether the generated schema is an accurate representation of the instance data.

If elements or attributes in more than one namespace are present, MapForce generates a separate XML Schema for each distinct namespace; therefore, multiple files may be created on the disk.

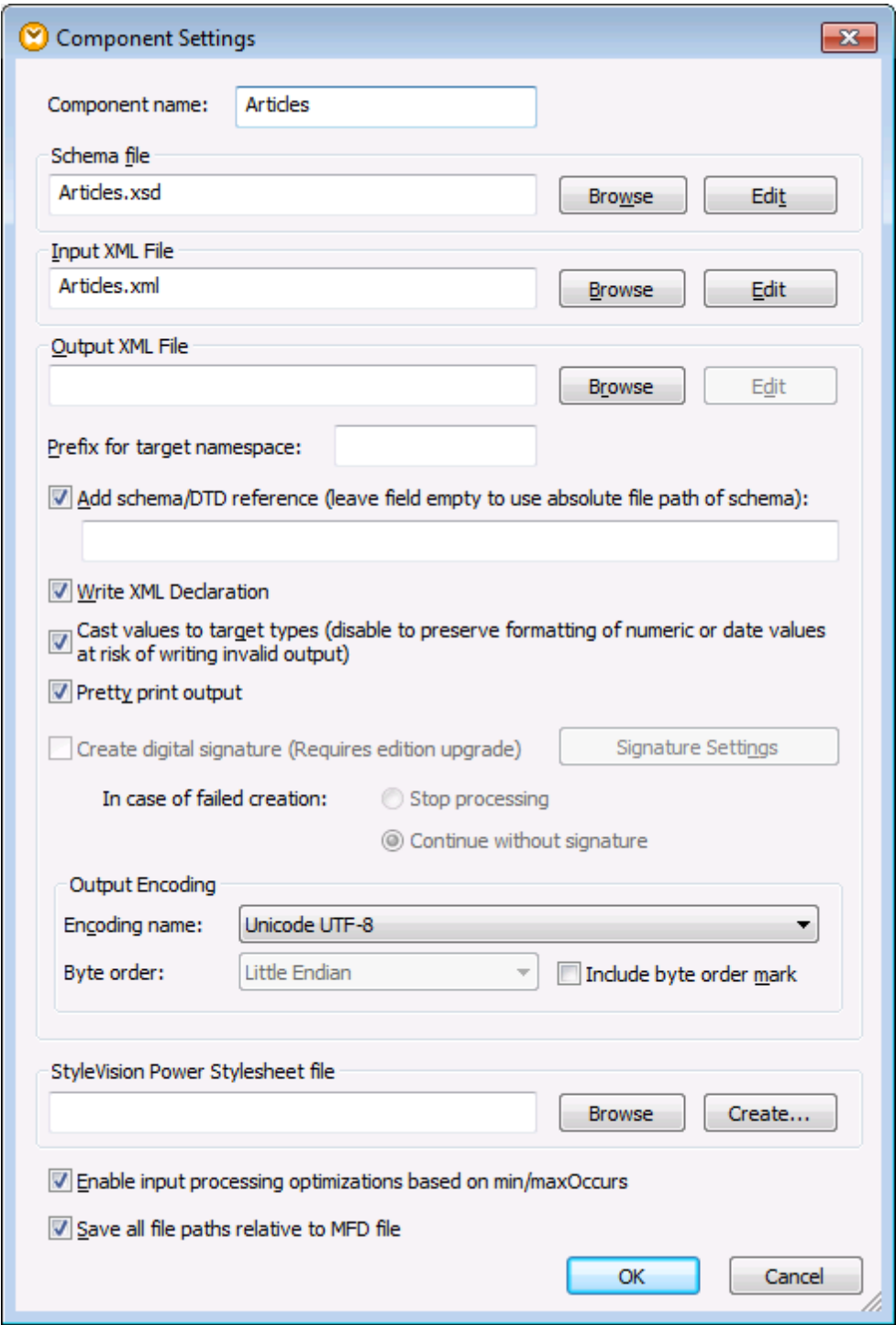


### 6.1.2 XML Component Settings

After you add an XML component to the mapping area, you can configure the settings applicable to it from the Component Settings dialog box. You can open the Component settings dialog box in one of the following ways:

- Select the component on the mapping, and, on the **Component** menu, click **Properties**.
- Double-click the component header.
- Right-click the component header, and then click **Properties**.





XML Component Settings dialog box

The available settings are as follows.

Component name	The component name is automatically generated when you
----------------	--------------------------------------------------------



	<p>create the component. You can however change the name at any time.</p> <p>If the component name was automatically generated and you select an instance file after that, MapForce will prompt you to optionally update the component name as well.</p> <p>The component name can contain spaces (for example, "Source XML File") and full stop characters (for example, "Orders.EDI"). The component name may not contain slashes, backslashes, colons, double quotes, leading or trailing spaces. In general, be aware of the following implications when changing the name of the component:</p> <ul style="list-style-type: none"> <li>• If you intend to deploy the mapping to FlowForce Server, the component name must be unique.</li> <li>• It is recommended to use only characters that can be entered at the command line. National characters may have different encodings in Windows and at the command line.</li> </ul>
<i>Schema file</i>	<p>Specifies the name or path of the XML schema file used by MapForce to validate and map data.</p> <p>To change the schema file, click <b>Browse</b> and select the new file. To edit the file in XMLSpy, click <b>Edit</b>.</p>
<i>Input XML file</i>	<p>Specifies the XML instance file from which MapForce will read data. This field is meaningful for a source component and is filled when you first create the component and assign to it an XML instance file.</p> <p>In a source component, the instance file name is also used to detect the XML root element and the referenced schema, and to validate against the selected schema.</p> <p>To change the location of the file, click <b>Browse</b> and select the new file. To edit the file in XMLSpy, click <b>Edit</b>.</p>
<i>Output XML file</i>	<p>Specifies the XML instance file to which MapForce will write data. This field is meaningful for a target component.</p> <p>To change the location of the file, click <b>Browse</b> and select the new file. To edit the file in XMLSpy, click <b>Edit</b>.</p>
<i>Prefix for target namespace</i>	<p>Allows you to enter a prefix for the target namespace. Ensure that the target namespace is defined in the target schema, before assigning the prefix.</p>
<i>Add schema/DTD reference</i>	<p>Adds the path of the referenced XML Schema file to the root element of the XML output. The path of the schema entered in this field is written into the generated target instance files in the <code>xsi:schemaLocation</code> attribute, or into the <code>DOCTYPE</code></p>



	<p>declaration if a DTD is used.</p> <p>Entering a path in this field allows you to define where the schema file referenced by the XML instance file is to be located. This ensures that the output instance can be validated at the mapping destination when the mapping is executed. You can enter an <b>http://</b> address as well as an absolute or relative path in this field.</p> <p>Deactivating this option allows you to decouple the XML instance from the referenced XML Schema or DTD (for example, if you want to send the resulting XML output to someone who does not have access to the underlying XML Schema).</p>						
<i>Write XML declaration</i>	<p>This option enables you to suppress the XML declaration from the generated output. By default, the option is enabled, meaning that the XML declaration is written to the output.</p> <p>This feature is supported as follows in MapForce target languages and execution engines.</p> <table><tr><th>Target language / Execution engine</th><th>When output is a file</th><th>When output is a string</th></tr><tr><td>XSLT, XQuery</td><td>Yes</td><td>No</td></tr></table>	Target language / Execution engine	When output is a file	When output is a string	XSLT, XQuery	Yes	No
Target language / Execution engine	When output is a file	When output is a string					
XSLT, XQuery	Yes	No					
<i>Cast values to target types</i>	<p>Allows you to define if the target XML schema types should be used when mapping, or if all data mapped to the target component should be treated as <b>string</b> values. By default, this setting is enabled.</p> <p>Deactivating this option allows you to retain the precise formatting of values. For example, this is useful to satisfy a pattern facet in a schema that requires a specific number of decimal digits in a numeric value.</p> <p>You can use mapping functions to format the number as a string in the required format, and then map this string to the target.</p> <p>Note that disabling this option will also disable the detection of invalid values, e.g. writing letters into numeric fields.</p>						
<i>Pretty print output</i>	<p>Reformats the output XML document to give it a structured look. Each child node is offset from its parent by a single tab character.</p>						
<i>Output Encoding</i>	<p>Allows you specify the following settings of the output instance file:</p>						



	<ul style="list-style-type: none"> <li>• Encoding name</li> <li>• Byte order</li> <li>• Whether the byte order mark (BOM) character should be included.</li> </ul> <p>By default, any new components have the encoding defined in the <b>Default encoding for new components</b> option. You can access this option from <b>Tools   Options</b>, General tab.</p> <p>If the mapping generates XSLT 1.0/2.0, activating the <b>Byte Order Mark</b> check box does not have any effect, as these languages do not support Byte Order Marks.</p>
<i>StyleVision Power Stylesheet file</i>	<p>This option allows you to select or create an Altova StyleVision stylesheet file. Such files enable you to output data from the XML instance file to a variety of formats suitable for reporting, such as HTML, RTF, and others.</p> <p>See also <a href="#">Using Relative Paths on a Component</a>.</p>
<i>Enable input processing optimizations based on min/maxOccurs</i>	<p>This option allows special handling for sequences that are known to contain exactly one item, such as required attributes or child elements with <code>minOccurs</code> and <code>maxOccurs="1"</code>. In this case, the first item of the sequence is extracted, then the item is directly processed as an atomic value (and not as a sequence).</p> <p>If the input data is <b>not valid</b> against the schema, an empty sequence might be encountered in a mapping, which stops the mapping with an error message. To allow the processing of such <b>invalid input</b>, disable this check box.</p>
<i>Save all file paths relative to MFD file</i>	<p>When this option is enabled, MapForce saves the file paths displayed on the Component Settings dialog box relative to the location of the MapForce Design (.mfd) file. See also <a href="#">Using Relative Paths on a Component</a>.</p>

### 6.1.3 Using DTDs as "Schema" Components

Starting with MapForce 2006 SP2, namespace-aware DTDs are supported for source and target components. The namespace-URLs are extracted from the DTD "xmlns"-attribute declarations, to make mappings possible.

However, some DTDs contain xmlns\*-attribute declarations without namespace-URLs (for example, DTDs used by StyleVision ). Such DTDs have to be extended to make them useable in MapForce. Specifically, you can make such DTDs useable by defining the xmlns-attribute with the namespace-URI, as shown below:

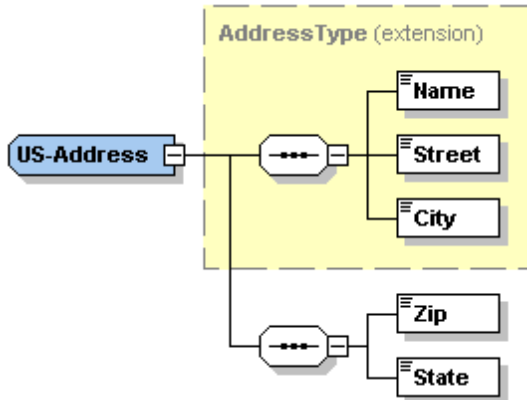


```
<!ATTLIST fo:root
  xmlns:fo CDATA #FIXED 'http://www.w3.org/1999/XSL/Format'
  ...
>
```

### 6.1.4 Derived XML Schema Types

MapForce supports the mapping to/from derived types of a complex type. Derived types are complex types of an XML Schema that use the **xsi:type** attribute to identify the specific derived types.

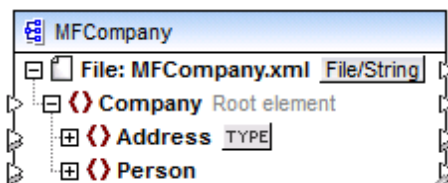
The screenshot below shows the definition of a derived type called `US-Address`, in XMLSpy. The base type (or originating complex type) is `AddressType`. Two extra elements were added to create the derived type `US-Address`: `Zip` and `State`.



Sample derived type (XMLSpy schema view)

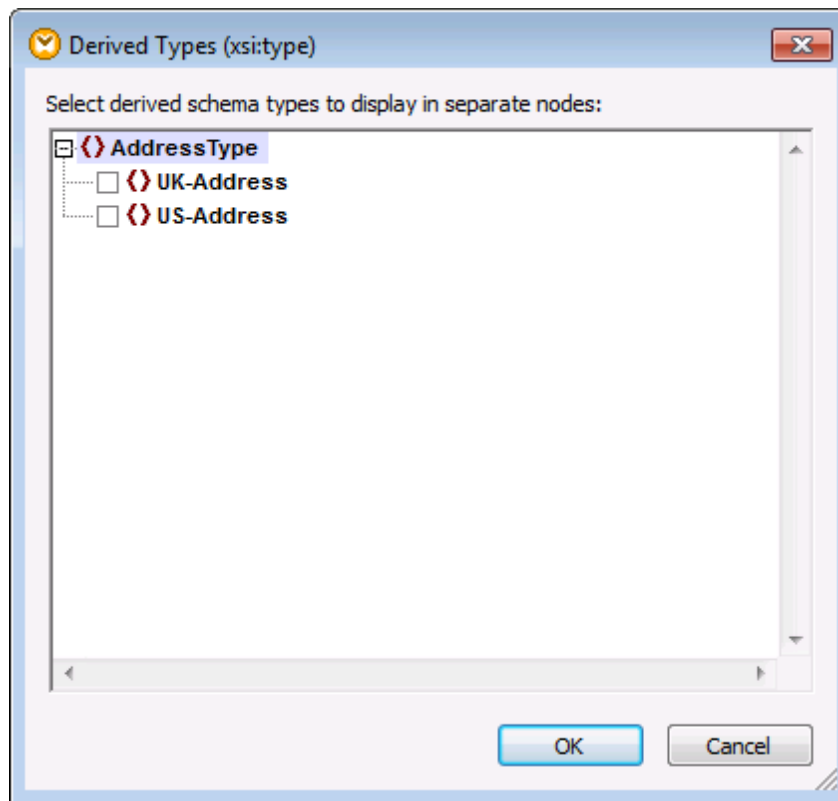
The following example shows you how to map data to or from derived XML schema types.

1. On the **Insert** menu, click **XML Schema/File**, and open the following XML Schema: **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\MFCompany.xsd**.
2. When prompted to supply an instance file, click **Skip**, and then select `Company` as the root element.

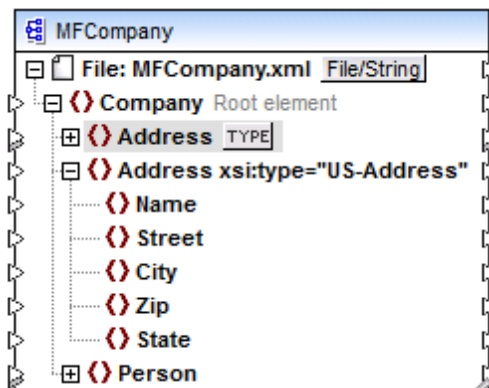


3. Click the `TYPE` button next to the `Address` element. This button indicates that derived types exist for this element in the schema.





4. Select the check box next to the derived type you want to use (US-Address, in this case), and confirm with OK. A new element `Address xsi:type="US-Address"` has been added to the component.



You can now map data to or from the `US-Address` derived type.

Note that you can also include multiple derived types by selecting them in the Derived Types dialog box. In this case, each would have its own `xsi:type` element in the component.



### 6.1.5 QNames

MapForce resolves QName (qualified name) prefixes (<https://www.w3.org/TR/xml-names/#ns-qualnames>) when reading data from XML files at mapping execution run-time.

QNames are used to reference and abbreviate namespace URLs in XML instance documents. There are two types of QNames: Prefixed and Unprefixed QNames.

PrefixedName      Prefix ':' LocalPart

UnPrefixedName                  LocalPart  
where LocalPart is an Element or Attribute name.

For example, in the listing below, `<x:p/>` is a QName, where:

- the prefix "x" is an abbreviation of the namespace "<http://myCompany.com>".
- p is the local part.

```
<?xml version='1.0'?>
<doc xmlns:x="http://myCompany.com">
  <x:p/>
</doc>
```

MapForce also includes several QName-related functions in the [core | QName functions](#) library.

### 6.1.6 Nil Values / Nillable

The XML Schema specification allows for an element to be valid without content if the `nillable="true"` attribute has been defined for that specific element in the schema. In the instance XML document, you can then indicate that the value of an element is nil by adding the `xsi:nil="true"` attribute to it. This section describes how MapForce handles nil elements in source and target components.

#### 'xsi:nil' versus 'nillable'

The `xsi:nil="true"` attribute is defined in the XML **instance** document.

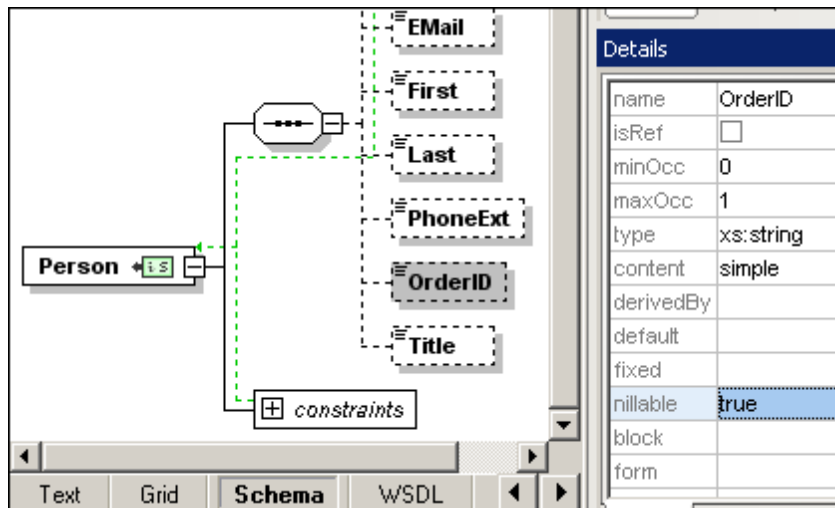
```
14  <Person>
15    <PrimaryKey>2</PrimaryKey>
16    <ForeignKey>1</ForeignKey>
17    <EMail>biff@amail.com</EMail>
18    <First>biff</First>
19    <Last>bander</Last>
20    <PhoneExt>22</PhoneExt>
21    <OrderID xsi:nil="true"/>
22    <Title>IT services</Title>
23  </Person>
```



The `xsi:nil="true"` attribute indicates that, although the element exists, it has no content. Note that the `xsi:nil="true"` attribute applies to element values, and not to attribute values. An element with `xsi:nil="true"` may still have other attributes, even if it does not have content.

The `xsi:nil` attribute is not displayed explicitly in the MapForce graphical mapping, because it is handled automatically in most cases. Specifically, a "nilled" node (one that has the `xsi:nil="true"` attribute) exists, but its content does not exist.

The `nillable="true"` attribute is defined in the XML **schema**. In MapForce, it can be present in both the source and target components.



### Nillable elements as mapping source

MapForce checks the `xsi:nil` attribute automatically, whenever a mapping reads data from nilled XML elements. If the value of `xsi:nil` is `true`, the content will be treated as non-existent.

When you create a **Target-driven** mapping from a nillable source element to a nillable target element with **simple content** (a single value with optional attributes, but without child elements), where `xsi:nil` is set on a source element, MapForce adds the `xsi:nil` attribute to the target element (for example, `<OrderID xsi:nil="true"/>`).

When you create a **Copy-All** mapping from a nillable source element to a nillable target element, where `xsi:nil` is set on a source element, MapForce adds the `xsi:nil` attribute to the target element (for example, `<OrderID xsi:nil="true"/>`).

To check explicitly whether a source element has the `xsi:nil` attribute set to `true`, use the [is-xsi-nil](#) function. It returns `TRUE` for nilled elements and `FALSE` for other nodes.

To substitute a nilled (non-existing) source element value with something specific, use the [substitute-missing](#) function.

#### Notes:

- Connecting the [exists](#) function to a nilled source element returns `TRUE`, since the element node actually exists, even if it has no content.



- Using functions that expect simple values (such as **multiply** and **concat**) on elements where `xsi:nil` has been set does not yield a result, as no element content is present and no value can be extracted. These functions behave as if the source node did not exist.

### Nillable elements as mapping target

When you create a **Target-driven** mapping from a nillable source element to a nillable target element with **simple content** (a single value with optional additional attributes, but without child elements), where `xsi:nil` is set on a source element, MapForce inserts the `xsi:nil` attribute into the target element (for example, `<OrderID xsi:nil="true"/>`). If the `xsi:nil="true"` attribute has not been set in the XML source element, then the element content is mapped to the target element in the usual fashion.

When mapping to a nillable target element with **complex type** (with child elements), the `xsi:nil` attribute will **not** be written automatically, because MapForce cannot know at the time of writing the element's attributes if any child elements will follow. For such cases, define a **Copy-All** connection to copy the `xsi:nil` attribute from the source element.

When mapping an **empty sequence** to a target element, the element will not be created at all, independent of its nillable designation.

To force the creation of an empty target element with `xsi:nil` set to `true`, connect the [set-xsi-nil](#) function directly to the target element. This works for target elements with simple and complex types.

If the node has simple type, use the [substitute-missing-with-xsi-nil](#) function to insert `xsi:nil` in the target if no value from your mapping source is available. This can happen if the source node does not exist at all, or if a calculation (for example, multiply) involved a nilled source node and therefore yielded no result.

#### Note:

- Functions which generate `xsi:nil` cannot be passed through functions or components which only operate on values (such as the **if-else** function).

## 6.1.7 Comments and Processing Instructions

Comments and Processing Instructions can be inserted into target XML components. Processing instructions are used to pass information to applications that further process XML documents. Note that Comments and Processing instructions cannot be defined for nodes that are part of a copy-all mapped group.

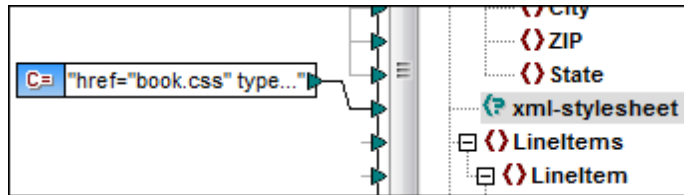
### To insert a Processing Instruction:

1. Right-click an element in the target component and select Comment/Processing Instruction, then one of the Processing Instruction options from the menu (Before, After)
2. Enter the Processing Instruction (target) name in the dialog and press OK to confirm, e.g.



xml-stylesheet.

This adds a node of this name to the component tree.



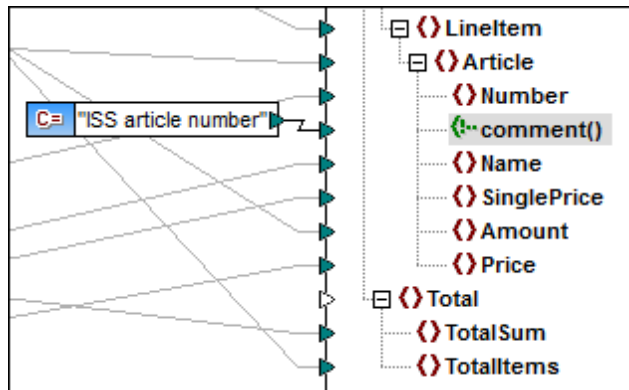
3. You can use, for example, a constant component to supply the value of the Processing Instruction attribute, e.g. `href="book.css" type="text/css"`.

Note:

Multiple Processing Instructions can be added before or after any element in the target component.

#### To insert a comment:

1. Right-click an element in the target component and select Comment/Processing Instruction, then one of the Comment options from the menu (Before, After).



This adds the comment node ( `<!--comment()>` ) to the component tree.

2. Use a constant component to supply the comment text, or connect a source node to the comment node.

Note:

Only one comment can be added before and after a single target node. To create multiple comments, use the duplicate input function.

#### To delete a Comment/Processing Instruction:

- Right-click the respective node, select Comment/Processing Instruction, then select Delete Comment/Processing Instruction from the flyout menu.



### 6.1.8 CDATA Sections

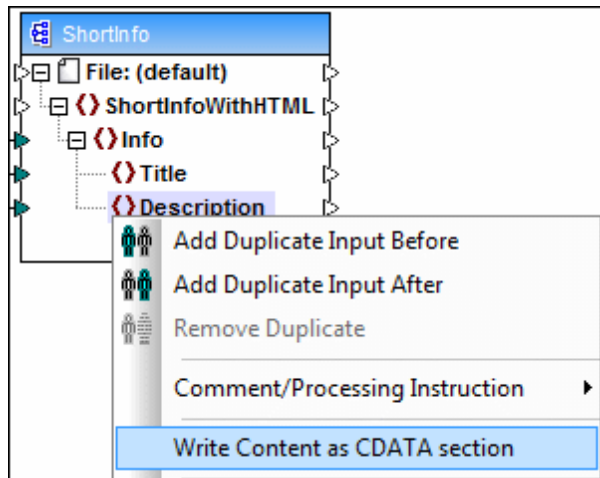
CDATA sections are used to escape blocks of text containing characters which would normally be interpreted as markup. CDATA sections start with "<![CDATA[" and end with the "]]>".

Target nodes can now write the input data that they receive as CDATA sections. The target node components can be:

- XML data
- XML data embedded in database fields
- XML child elements of typed dimensions in an XBRL target

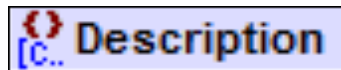
**To create a CDATA section:**

1. Right-click the target node that you want to define as the CDATA section and select "Write Content as CDATA section".



A prompt appears warning you that the input data should not contain the CDATA section close delimiter ']]>', click OK to close the prompt.

The [C.. icon shown below the element tag shows that this node is now defined as a CDATA section.



Note:

CDATA sections can also be defined on duplicate nodes, and xsi:type nodes.

Example:

The **HTMLinCDATA.mfd** mapping file available in the ...\\MapForceExamples folder shows an example of where CDATA sections can be very useful.

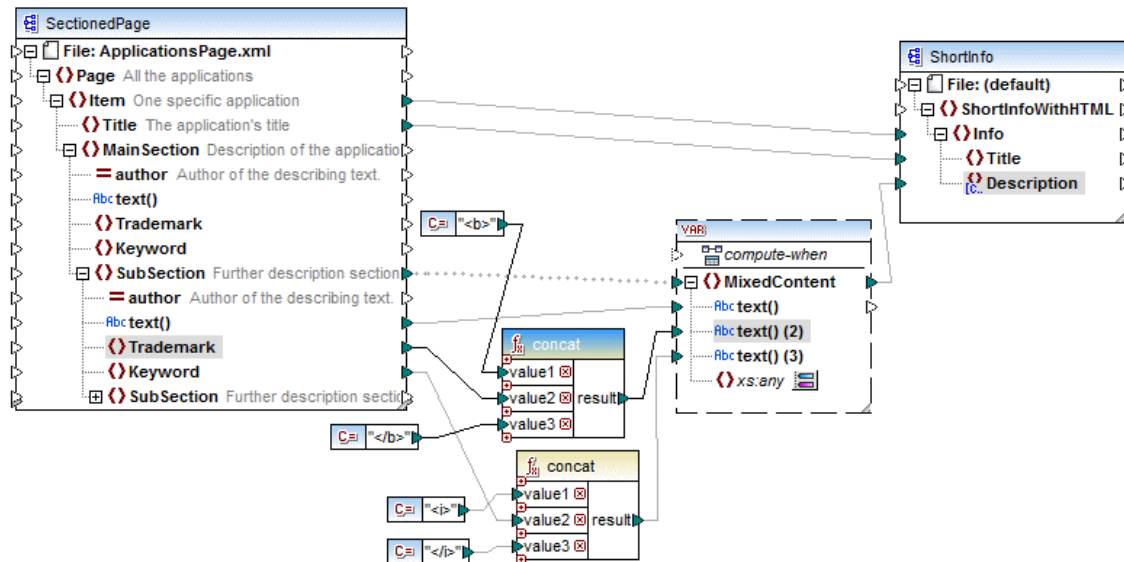
In this example:

- Bold start (<b>) and end (</b>) tags are added to the content of the **Trademark** source element.
- Italic start (<i>) and end (</i>) tags are added to the content of the **Keyword** source



element.

- The resulting data is passed on to duplicate **text()** nodes in the order that they appear in the source document, due to the fact the Subsection element connector, has been defined as a [Source Driven](#) (Mixed content) node.
- The output of the MixedContent node is then passed on to the **Description** node in the ShortInfo target component, which has been defined as a CDATA section.



Clicking the Output button shows the CDATA section containing the marked-up text.

```

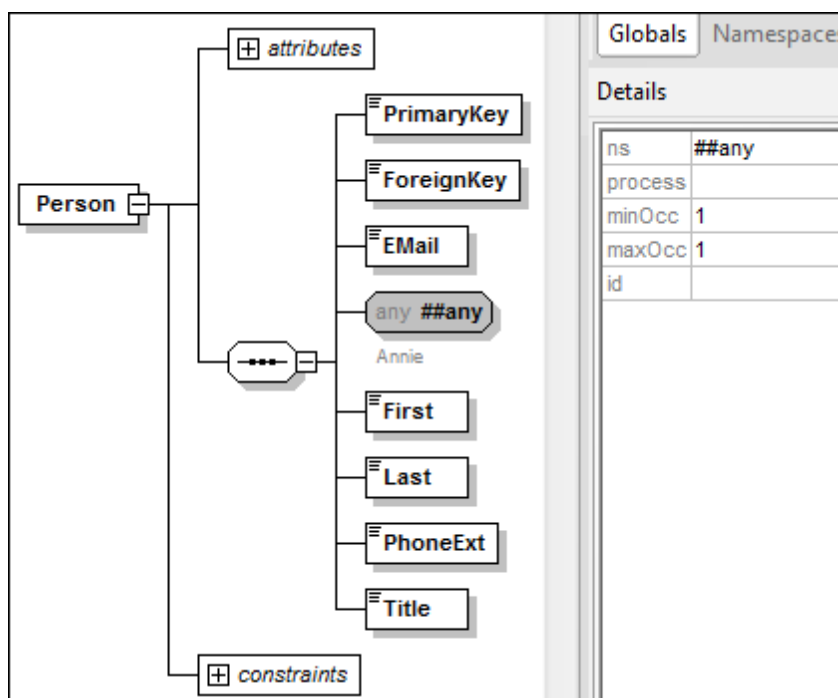
7  <Info>
8    <Title>MapForce</Title>
9    <Description><![CDATA[Altova <b>MapForce</b> 2014 Enterprise Edition is the premier <i>XML</i>
10   / <i>database</i> / <i>flat file</i> / <i>EDI</i> data mapping tool that auto-generates mapping code in
    <i>XSLT</i> 1.0/2.0, <i>XQuery</i>, <i>Java</i>, <i>C++</i> and <i>C#</i>. It is the definitive tool for
    data integration and information leverage.]]></Description>
  </Info>


```

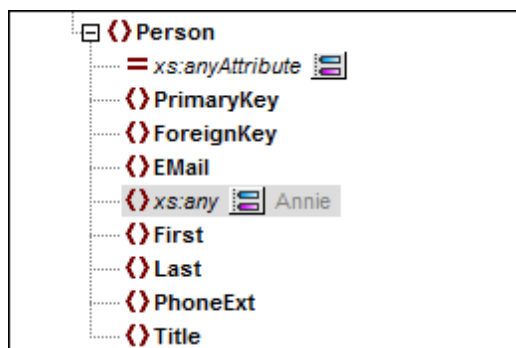
### 6.1.9 Wildcards - xs:any / xs:anyAttribute


The wildcards `xs:any` (and `xs:anyAttribute`) allow you to use any elements/attributes from schemas. The screenshot shows the "any" element in the Schema view of XMLSpy.



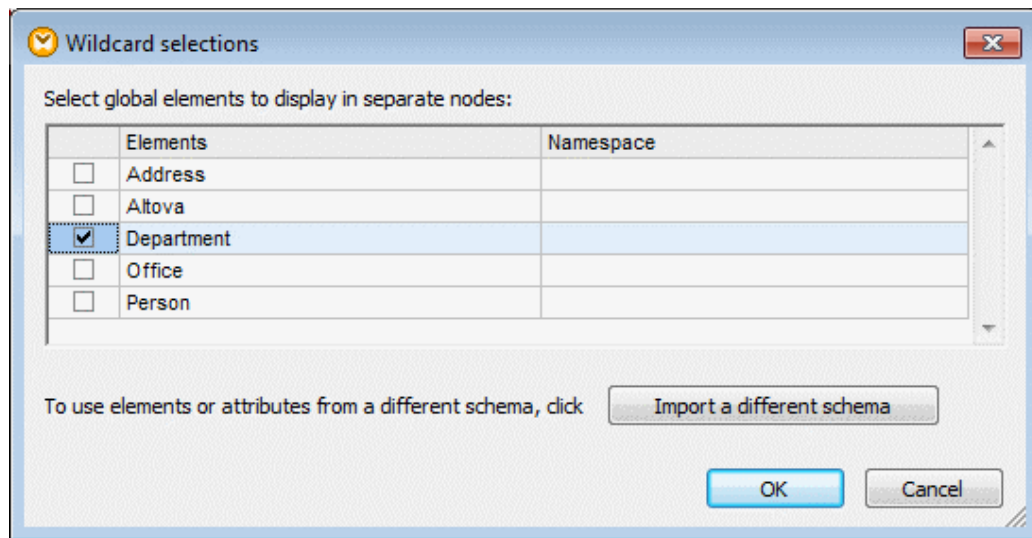



In MapForce, a **Change Selection** (  ) button appears to the right of the `xs:any` element (or `xs:anyAttribute`).

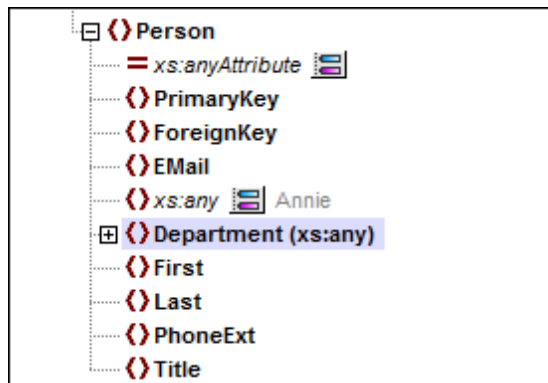


When clicked, the **Change Selection** button  opens the "Wildcard selections" dialog box. The entries in this list show the global elements and attributes declared in the current schema.





Clicking one or more of the check boxes and confirming with OK, inserts that element/attribute (and any other child nodes) into the component. The wildcard elements or attributes are inserted immediately after the node whose **Change Selection** (  ) button was clicked.



You can now map to/from these nodes as with any other element.

On a component, the wildcard elements or attributes can be recognized by the **(xs:any)** text appended to their name.

To remove a wildcard element, click the Change Selection (  ) button, and then deselect it from the "Wildcard selections" dialog box.

### Wildcards and dynamic node names

Mapping data to or from wildcards is generally suitable where all possible elements or attributes that appear in the XML instance are declared by the component's XML schema (or can be imported from external schemas). However, there may be situations where elements or attributes appearing in an instance are too many to be declared in the schema. Consider the following instance where the number of child elements of `<message>` is arbitrary:



```
<?xml version="1.0" encoding="UTF-8"?>
<message>
  <line1>1</line1>
  <line2>2</line2>
  <line3>3</line3>
  .....
  <line999></line999>
</message>
```

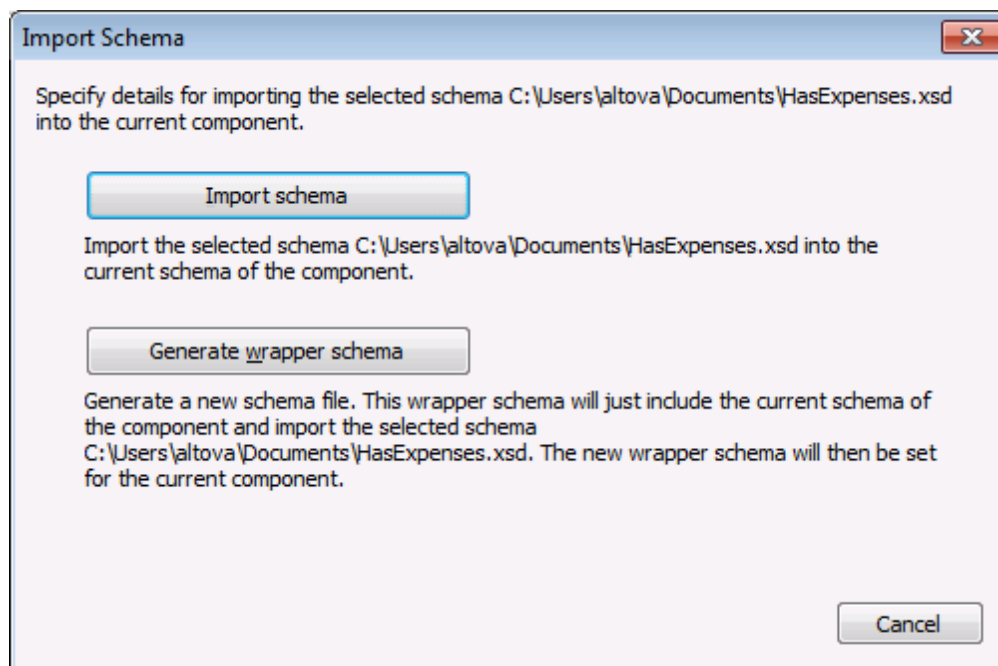
For such situations, use dynamic access to node names (see [Mapping Node Names](#)) instead of wildcards.

### Adding elements from a different schema as wildcards

Elements from a schema other than the one assigned to the component can also be used as wildcards. To make such elements visible on the component, click the **Import a different schema** button on the "Wildcard selections" dialog box. This opens a new dialog box where you have two options:

1. Import schema
2. Generate wrapper schema

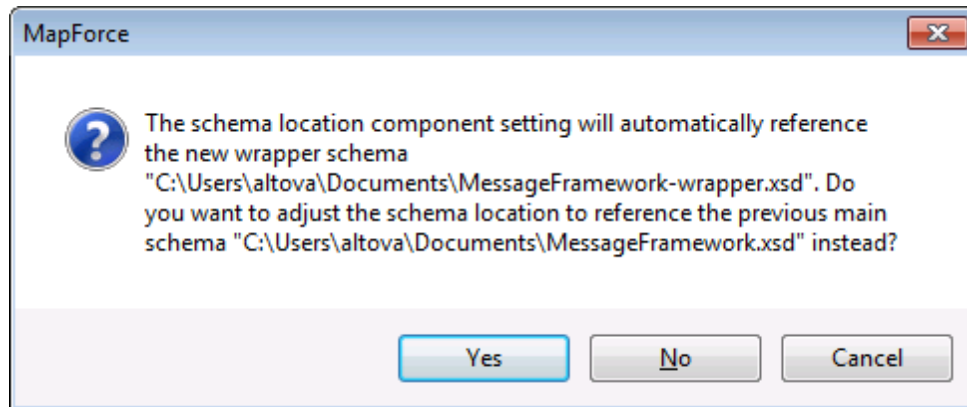
For example, the image below illustrates what happens if you attempt to import an external schema called **HasExpenses.xsd** into a current schema assigned to a component.



The **Import schema** option imports the external schema into the current schema assigned to the component. Be aware that this option overrides the existing schema of the component on the disk. If the current schema is a remote schema that was opened from a URL (see [Adding Components from a URL](#)) and not from the disk, it cannot be modified. In this case, use the **Generate wrapper schema** option.



The **Generate wrapper schema** option creates a new schema file called a "wrapper" schema. The advantage of using this option is that the existing schema of the component is not modified. Instead, a new schema will be created (that is, the wrapper schema) which will include both the existing schema and the schema to be imported. When you select this option, you are prompted to choose where the wrapper schema should be saved. By default, the wrapper schema has a name in the form **somefile-wrapper.xsd**. After you save the wrapper schema, it is by default automatically assigned to the component, and a dialog box prompts you:



Click **Yes** to revert to the previous schema; otherwise click **No** to keep the newly created wrapper schema assigned to the component.

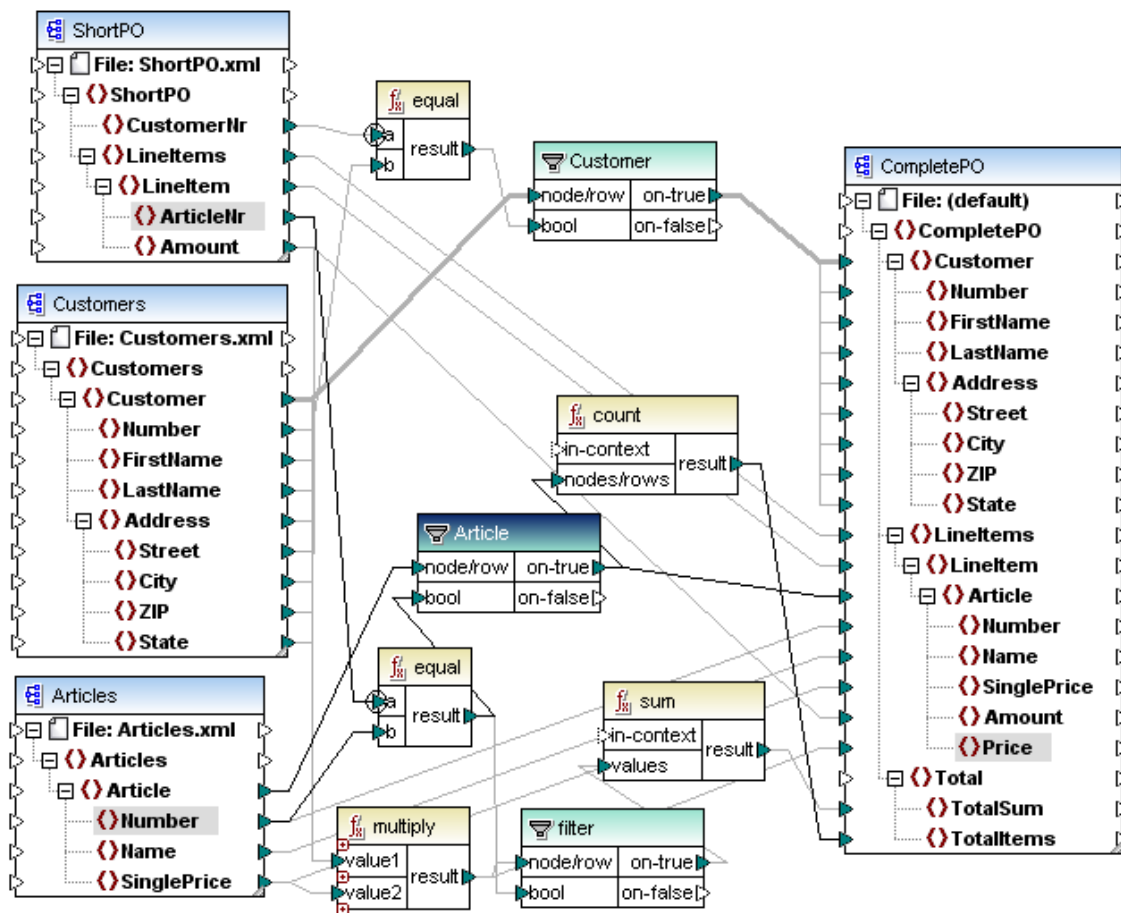
### 6.1.10 Merging Data from Multiple Schemas

MapForce allows you to merge multiple files into a single target file.

This example merges multiple source components with different schemas to a target schema. To merge an arbitrary number of files using the same schema, see [Processing Multiple Input or Output Files Dynamically](#).

The **CompletePO.mfd** file available in the [...\MapForceExamples](#) folder shows how three XML files are merged into one purchasing order XML file.





Note that multiple source component data are combined into one target XML file - CompletePO

- **ShortPO** is a schema with an associated XML instance file and contains only customer number and article data, i.e. Line item, number and amount. (There is only one customer in this file with the Customer number of 3)
- **Customers** is a schema with an associated XML instance file and contains customer number and customer information details, i.e. Name and Address info.
- **Articles** is a schema with an associated XML instance and contains article data, i.e. article name number and price.
- **CompletePO** is a schema file without an instance file as all the data is supplied by the three XML instance files. The hierarchical structure of this file makes it possible to merge and output all XML data.

This schema file has to be created in an XML editor such as XMLSpy, it is not generated by MapForce (although it would be possible to create if you had a **CompletePO.xml** instance file).

The structure of CompletePO is a combination of the source XML file structures.

The **filter** component (Customer) is used to find/filter the data where the customer numbers are identical in both the ShortPO and Customers XML files, and pass on the associated data to the target CompletePO component.

- The **CustomerNr** in ShortPO is compared with the **Number** in Customers using the "equal" function.
- As ShortPO only contains one customer (number 3), only customer and article data for



customer number 3, can be passed on to the filter component.

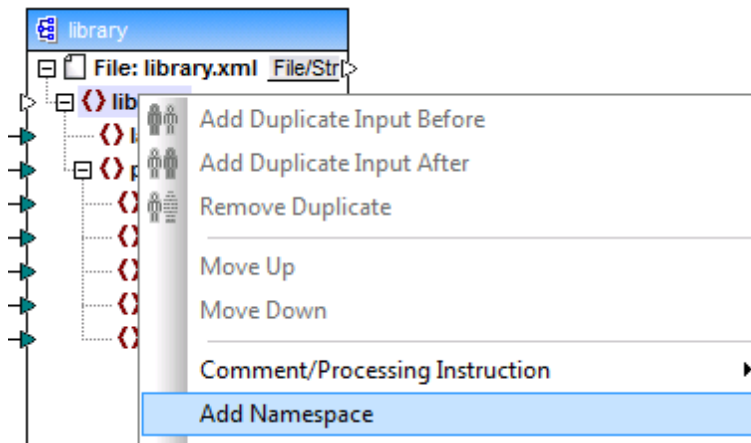
- The **node/row** parameter, of the filter component, passes on the **Customer** data to "on-true" when the bool parameter is true, i.e. when the same number has been found, in this case customer number 3.
- The rest of the customer and article data are passed on to the target schema through the two other filter components.

### 6.1.11 Declaring Custom Namespaces

By default, when a mapping produces XML output, the namespace (or set of namespaces) of each element and attribute is automatically derived by MapForce from the schema associated with the [target component](#). This is the default behavior in MapForce and is suitable for most mapping scenarios that involve generation of XML output.

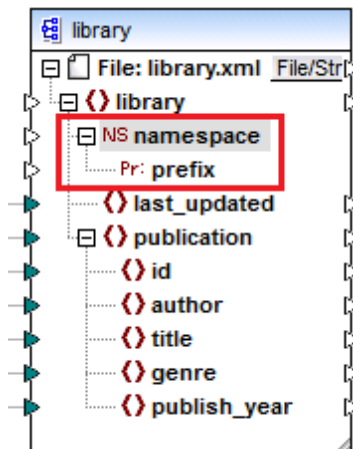
However, there might be cases when you want to have more control over namespaces of elements in the resulting XML output. For example, you may want to manually declare the namespace of an element directly from the mapping.

To understand how this works, open the **BooksToLibrary.mfd** mapping available in the **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\**. Right-click the `library` node, and select **Add Namespace** from the context menu.

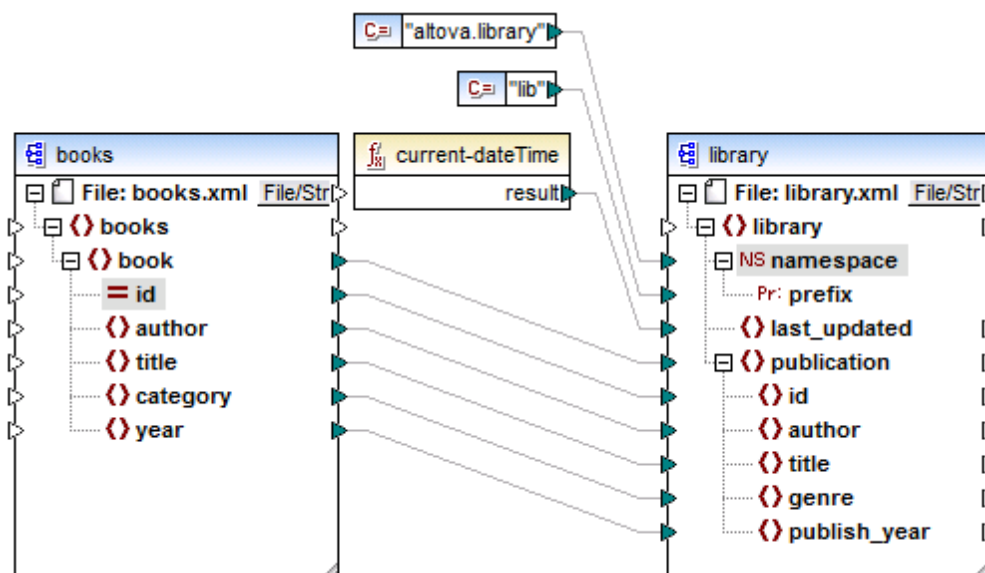


Notice that two new nodes are now available under the `library` node: a namespace and a prefix.





You can now map to them string values from the mapping. In the image below, two constants were defined (from **Insert | Constant** menu command) that provide the namespace "altova.library" and the prefix "lib":



The result is that, in the generated output, an `xmlns:<prefix>=<namespace>` attribute is added to the element, where `<prefix>` and `<namespace>` are values that come from the mapping (in this case, from constants). The generated output will now look as follows (notice the highlighted part):

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns:lib="altova.library" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:noNamespaceSchemaLocation="library.xsd">
...
```

**Note:** Declaring custom namespaces (and the **Add Namespace** command) is meaningful only for target XML components, and applies to elements only. The **Add Namespace**



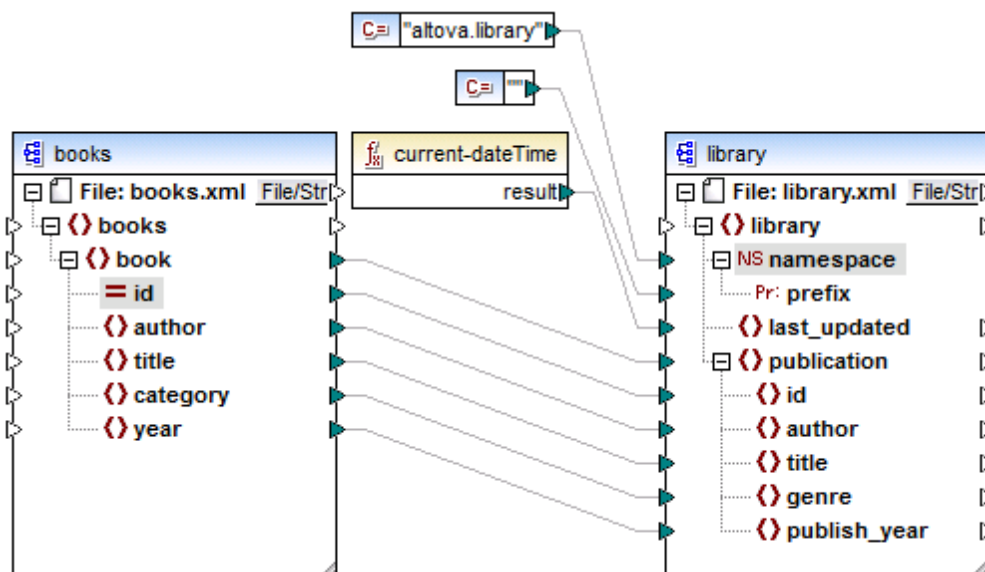
command is not available for attributes and wildcard nodes. It is also not available for nodes which receive data by means of a **Copy-All** connection.

You can also declare multiple namespaces for the same element, if necessary. To do this, right-click the node again, and select **Add Namespace** from the context menu. A new pair of namespace and prefix nodes become available, to which you can connect the new prefix and namespace values.

To remove a previously added namespace declaration, right-click the `ns:namespace` node, and select **Remove Namespace** from the context menu.

Both the `namespace` and `prefix` input connectors must be mapped, even if you provide empty values to them.

If you want to declare a default namespace (that is, one in the format `xmlns="mydefaultnamespace"`), map an empty string value to `prefix`. To see this case in action, edit the example mapping above so as to make the second constant an empty string.



The resulting output would then look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns="altova.library" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="library.xsd">
...
```

If you need to create prefixes for attribute names, for example `<number prod:id="prod557">557</number>`, you can achieve this by either enabling dynamic access to node's attributes (see [Mapping Node Names](#)), or by editing the schema so that it has a `prod:id` attribute for `<number>`.



## 6.2 HL7 Version 3

Support for HL7 version 3.x is automatically included in MapForce 2018 as it is XML based.

A separate installer for the HL7 V2.2 - V2.5.1 XML Schemas and configuration files is available on the Libraries page of the Altova website ( <https://www.altova.com/mapforce/download/libraries> )  
Select the Custom Setup in the installer, to only install the HL7 V3 components and XML Schemas.

Location of HL7 XML Schemas after installation:

32-bit MapForce on 32-bit operating system, or 64-bit MapForce on 64-bit operating system	C:\Program Files\Altova\Common2018\Schemas\hl7v3
32-bit MapForce on 64-bit operating system	C:\Program Files(x86)\Altova\Common2018\Schemas\hl7v3

HL7 documents can be used as source and target components in MapForce. This data can also be mapped to any number of XML schema components.







# Chapter 7

---

## Functions



## 7 Functions

Functions represent a powerful way to transform data according to your specific needs. This section provides instructions on working with functions (regardless if they are built-in to MapForce, defined by you, or reused from external sources). Use the following roadmap for quick access to specific tasks related to functions:

I want to...	Read this topic...
Add MapForce built-in functions or constants to the mapping	<ul style="list-style-type: none"><li>• <a href="#">Add a Built-in Function to the Mapping</a></li><li>• <a href="#">Add a Constant to the Mapping</a></li><li>• <a href="#">Search for a Function</a></li><li>• <a href="#">View a Function's Type and Description</a></li><li>• <a href="#">Add or Delete Function Arguments</a></li></ul>
Create my own functions in MapForce	<a href="#">User-Defined Functions</a>
Add custom XSLT functions to MapForce	<a href="#">Importing Custom XSLT 1.0 or 2.0 Functions</a>
View all built-in MapForce functions, or look up the description of a specific function.	<a href="#">Function Library Reference</a>

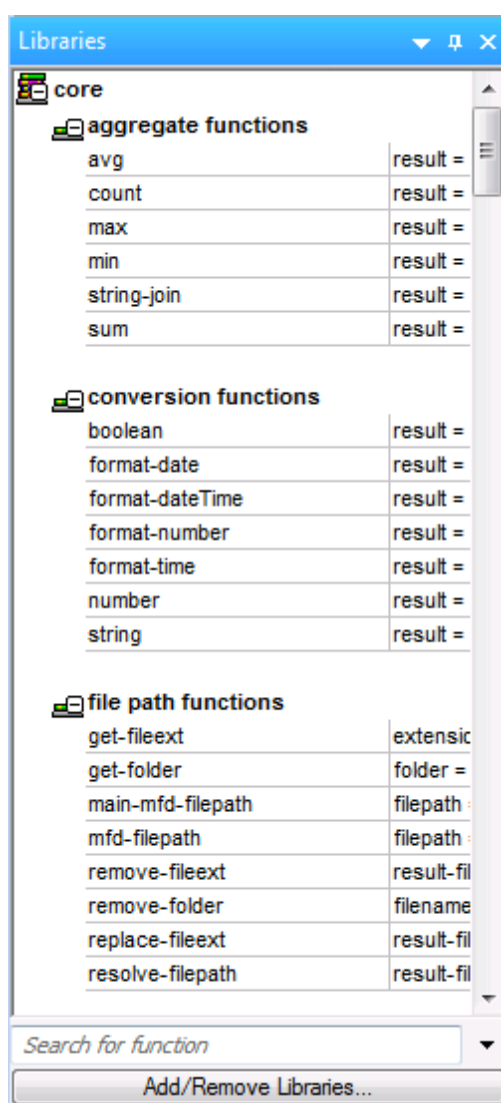


## 7.1 How To...

### 7.1.1 Add a Built-in Function to the Mapping

To use a function in a mapping:

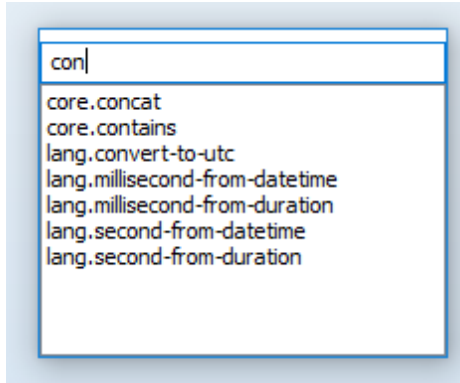
1. Select the transformation language (see [Selecting a transformation language](#)). Note that the list of available functions depends on the selected transformation language.
2. Click the required function in the Libraries window and drag it to the mapping area. To filter functions by name, start typing the function name in the text box located in the lower part of the window:



Alternatively, you can also quickly add a function to the mapping as follows:



1. Double-click anywhere on the empty area of the mapping and start typing the function name. A combo box appears with the same functions as in the Libraries window, filtered by the text you entered. To see a tooltip with more details about each function, select any function in the list.



2. Select the required function, and press **Enter** to add it to the mapping. To close the combo box without selecting a function, press **Escape**, or click anywhere outside the box.

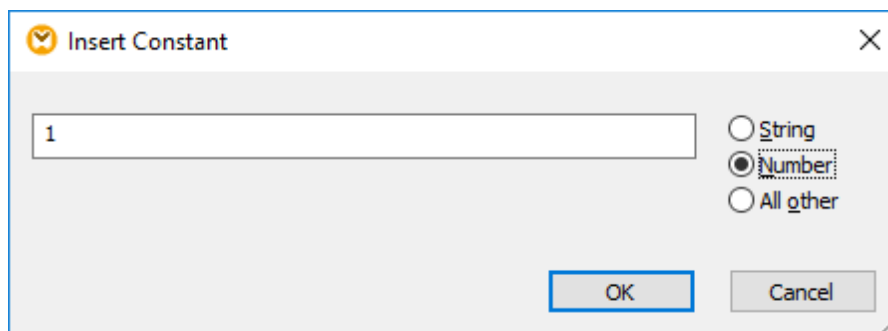
**Note:** Using the "double-click" alternative way described above, you can also add user-defined functions to the mapping.

## 7.1.2 Add a Constant to the Mapping

Constants enable you to supply custom text or numbers to the mapping. A constant's value, as the name implies, will remain the same for the duration of the mapping lifetime.

**To add a constant to the mapping:**

1. Do one of the following:
  - a. On the **Insert** menu, click **Constant**.
  - b. Right-click the mapping, and select **Insert Constant** from the context menu.



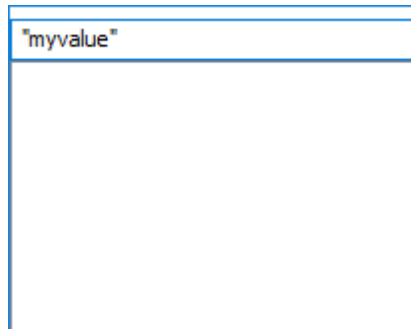
2. Enter the value of the constant, select the data type ("String", "Number", "All other"), and click **OK**.

Alternatively, you can also quickly add a constant as follows:

1. Double-click anywhere on an empty mapping area.



2. Do one of the following:
  - a. To add a string constant, start typing a double quote followed by the constant value.  
The closing double quote is optional.

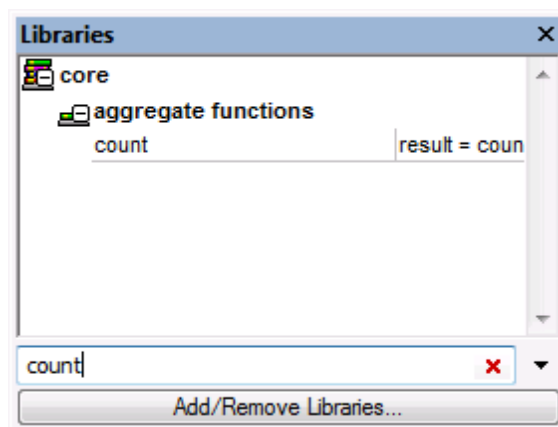


- b. To add a numeric constant, just type the number.
3. Press **Enter**.

### 7.1.3 Search for a Function

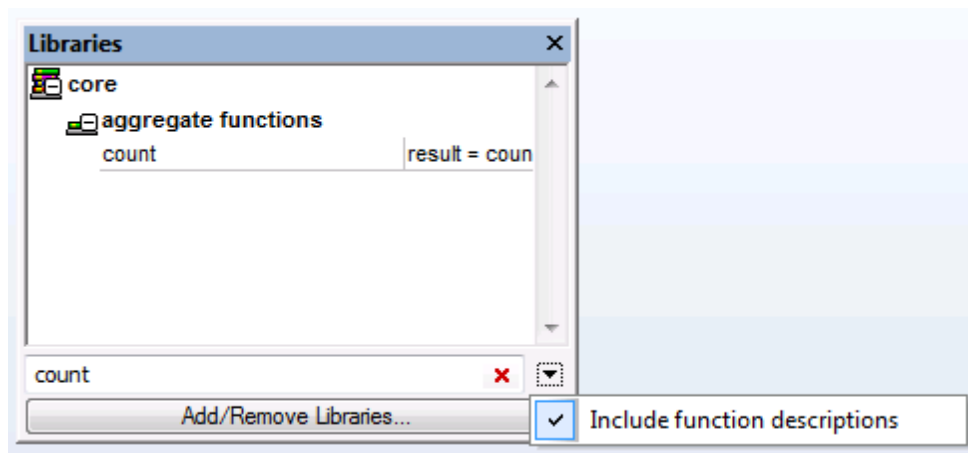
To search for a function in the Libraries window:

1. Start typing the function name in the text box located in the lower part of the Libraries window.



By default, MapForce searches by function name and description text. If you want to exclude the function description from the search, click the down-arrow and disable the **Include function descriptions** option.





To cancel the search, press the **Esc** key or click **X**.


The functions available in the Libraries window depend on the transformation language currently selected, see [Selecting a Transformation Language](#).

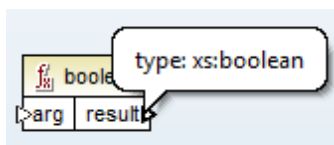
**To find all occurrences of a function within the currently active mapping:**

- Right-click the function name in the Libraries window, and select **Find All Calls** from the context menu. The search results are displayed in the Messages window.


## 7.1.4 View a Function's Type and Description

**To view the data type of a function input or output argument:**

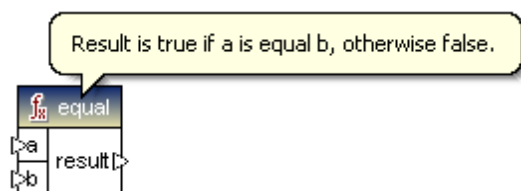
1. Make sure that the **Show tips**  toolbar button is enabled.
2. Move your mouse over the argument part of a function.



**To view the description of a function:**

1. Make sure that the **Show tips**  toolbar button is enabled.
2. Move your mouse of the function (this works both in the Libraries pane and on the mapping area)

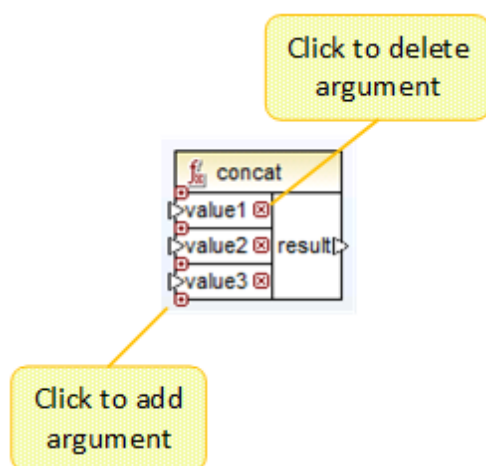





## 7.1.5 Add or Delete Function Arguments

To add or delete function arguments (for functions where that is applicable):

- Click **Add parameter** (  ) or **Delete parameter** (  ) next to the parameter you want to add or delete, respectively.



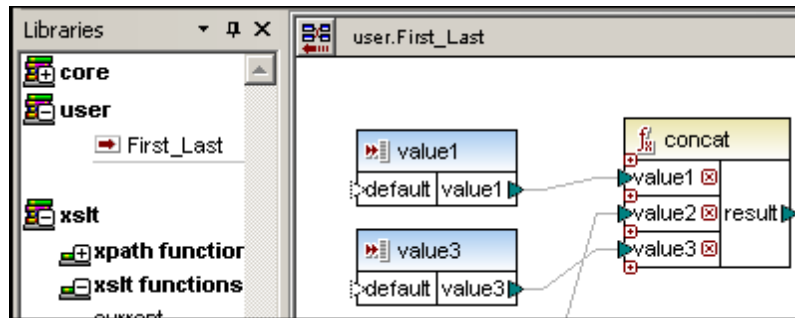
Dropping a connection on the  symbol automatically adds the parameter and connects it.



## 7.2 User-Defined Functions

MapForce allows you to create user-defined functions visually, in the same way as in the main mapping window.

These functions are then available as function entries in the Libraries window (for example, "First\_Last" in the image below), and are used in the same way as the currently existing functions. This allows you to organize your mapping into separate building blocks which are reusable across different mappings.



User-defined functions are stored in the \*.mfd file, along with the main mapping.

A user-defined function uses **input** and **output components** to pass information from the main mapping (or another user-defined function) to the user-defined function and back.

User-defined functions can contain "local" source components (i.e. that are within the user-defined function itself) such as XML schemas, which are useful when implementing lookup functions.

User-defined functions can contain any number of input and outputs where any of these can be in the form of: simple values, or XML nodes.

User-defined functions are useful when:

- combining multiple processing functions into a single component, e.g. for formatting a specific field or looking up a value
- reusing these components any number of times
- [importing](#) user-defined functions into other mappings (by loading the mapping file as a library)
- using [inline functions](#) to break down a complex mapping into smaller parts that can be edited individually
- mapping **recursive schemas** by creating [recursive user-defined functions](#)

User-defined functions can be either built from scratch, or from functions already available in the mapping tab.

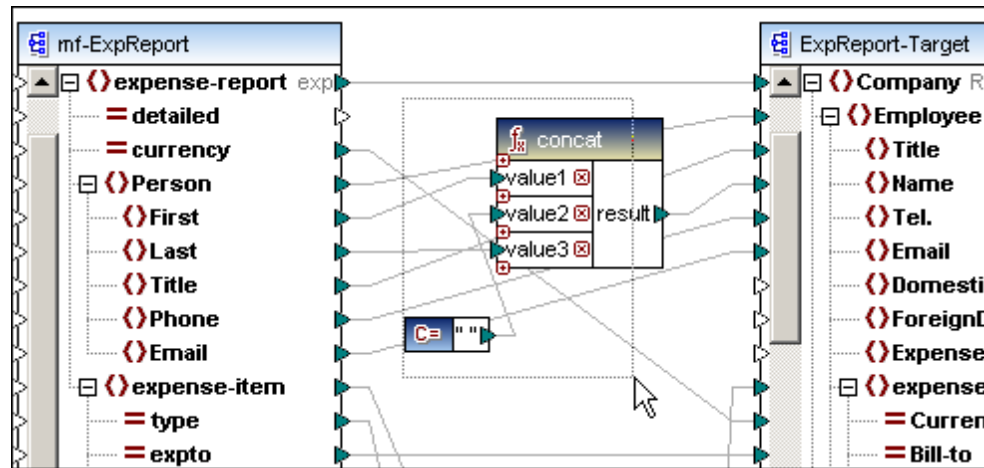
This example uses the **Tut-ExpReport.mfd** file available in the [...\MapForceExamples\Tutorial\](#) folder.

### Creating user-defined function from existing components

1. Drag to select both the "concat" and the constant components (you can also hold down



the CTRL key and click the functions individually).

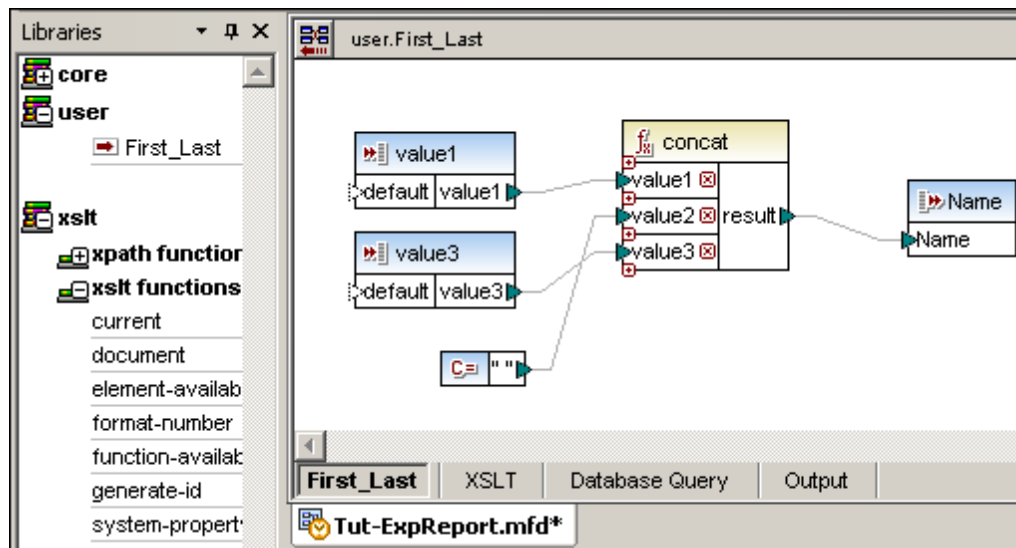



2. Select the menu option **Function | Create User-Defined Function from Selection**.
3. Enter the name of the new user-defined function (First\_Last).  
Note: valid characters are: alphanumeric, a-z, A-Z, 0-9 as well as underscore "\_", hyphen/dash "-" and colon ":".
4. Use the Syntax and Detail fields to add extra information on the new function, and click OK to confirm. The text you enter will appear as a tooltip when the cursor is placed over the function.

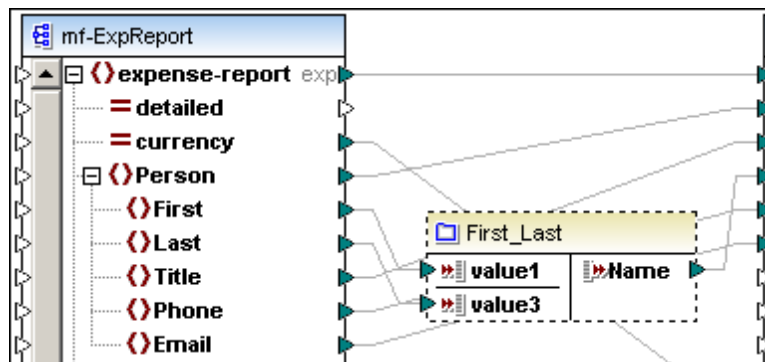
The library name "user" is supplied as a default, you can of course define your own library name in this field.

The individual elements that make up the function group appear in a tab with the function name. The new library "user" appears in the Libraries pane with the function name "First\_Last" below it.





Click the Home button  to return to the main mapping window. The components have now been combined into a single function component called First\_Last. The input and output parameters have been automatically connected.




Note that inline user-defined functions are displayed with a dashed outline. See [Inline user-defined functions](#) for more information.

Dragging the function name from the Libraries pane and dropping it in the mapping window, allows you to use it anywhere in the current mapping. To use it in a different mapping, please see [Reusing user-defined functions](#)

## Opening user-defined functions

To open a user-defined function, do one of the following:

- Double-click the title bar of a user-defined function component
- Double-click the specific user-defined function in the Libraries window.

This displays the individual components inside the function in a tab of that name. Click the Home button  to return to the main mapping. Double-clicking a user-defined function of a different \*.mfd file (in the main mapping window) opens that \*.mfd file in a new tab.



## Navigating user-defined functions

When navigating the various tabs (or user-defined function tabs) in MapForce, a history is automatically generated which allows you to travel forward or backward through the various tabs, by clicking the back/forward icons. The history is session-wide, allowing you to traverse multiple MFD files.



The Home button returns you to the main mapping tab from within the user-defined function.



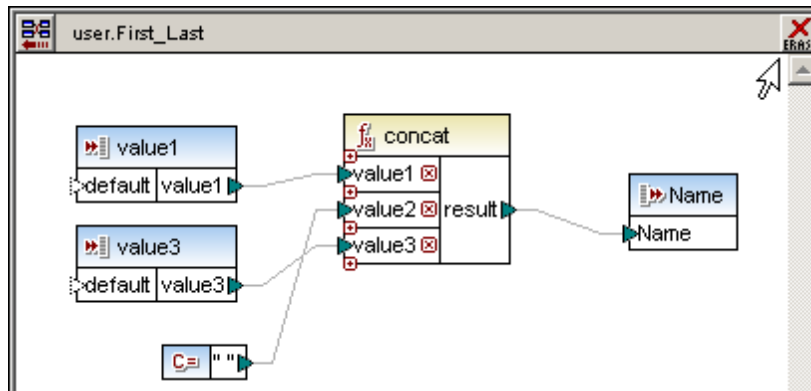
The Back button takes you back through your history



The Forward button moves you forward through your history

## Deleting user-defined functions from a library

1. Double-click the specific user-defined function in the Libraries window.
2. Click the **Erase** button in the top right of the title bar.



## Reusing (importing) user-defined functions

User-defined functions defined in one mapping can be imported into any other mapping as follows:

1. Click the **Add/Remove Libraries** button at the base of the Libraries window.
2. Click **Add** and select the \*.mfd file that contains the user-defined function(s) you want to import. The user-defined function now appears in the Libraries window. The library name is "user" if you created the user-defined function with the default library name. Otherwise, look for the library name that you specified when creating the user-defined function.
2. Drag the imported function from the Libraries window into the mapping.

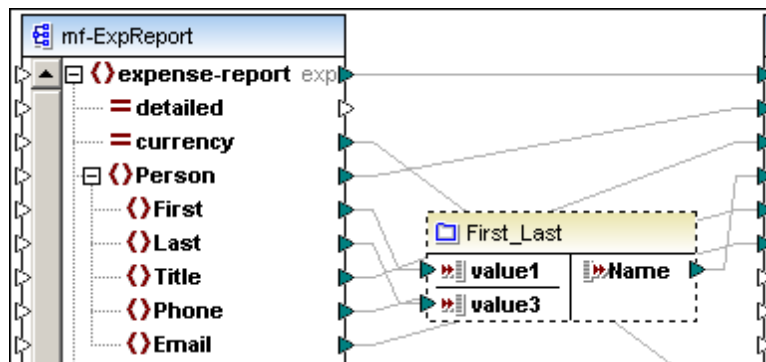
If the same library name is specified across multiple \*.mfd files, functions from all available sources appear under the same library name in the Libraries window. However, only the functions in the currently active document can be edited by



double-clicking.

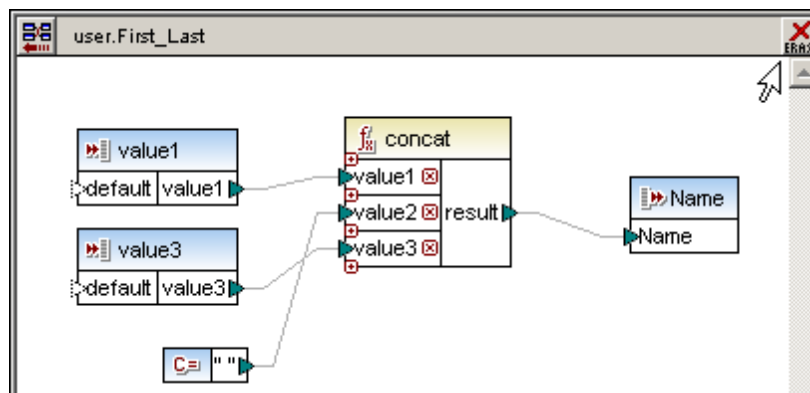
Note that possible changes in imported functions are applied to importing mappings when saving the library \*.mfd file.

### Parameter order in user-defined functions



The parameter order within user-defined functions can be directly influenced:

- Input and output parameters are sorted by their position from top to bottom (from the top left corner of the parameter component).
- If two parameters have the same vertical position, the leftmost takes precedence.
- In the unusual case that two parameters have exactly the same position, the internal component ID is automatically used.



Notes:

- The Component positioning and resizing actions are undoable.
- Newly added input or output components are created below the last input or output component.
- Complex and simple parameters can be mixed. The parameter order is derived from the component positions.

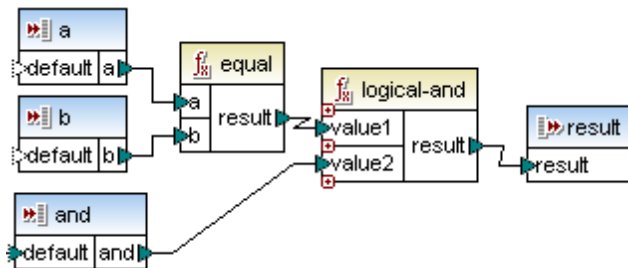


## 7.2.1 Function parameters

Function **parameters** are represented inside a user-defined function by **input** and **output components**.

Input components/parameters: **a, b, and**

Output component/parameter: **result**

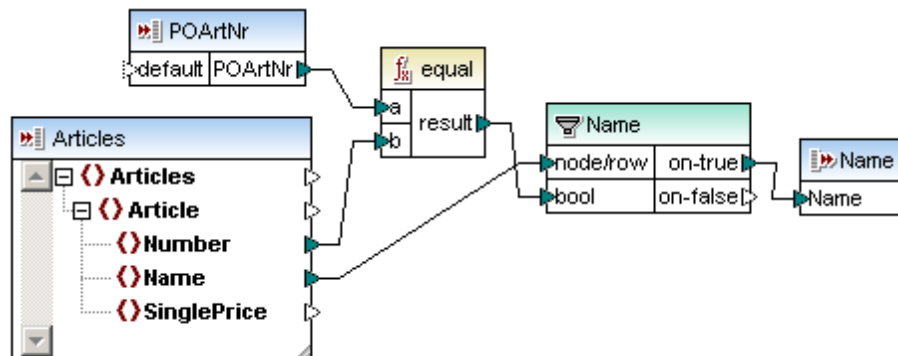


Input parameters are used to pass data from the main mapping into the user-defined function, while output parameters are used to return data back to the main mapping. Note that user-defined functions can also be called from other user-defined functions.

### Simple and complex parameters

The input and output parameters of user-defined functions can be of various types:

- Simple values, e.g. string or integer
- Complex node trees, e.g. an XML element with attributes and child nodes



Input parameter **POArtNr** is a simple value of datatype "string"

Input parameter **Articles** is a **complex** XML document node tree

Output parameter **Name** is a simple value of type string

Note:

The user-defined functions shown above are all available in the **PersonListByBranchOffice.mfd** file available in the ...\\MapForceExamples folder.

### Sequences

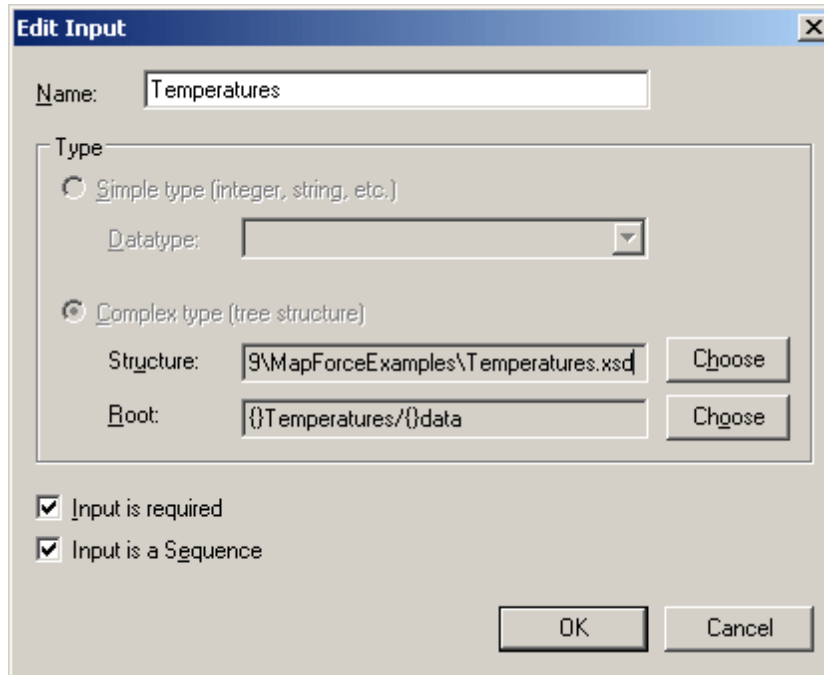
Sequences are data consisting of a range, or sequence, of values. Simple and complex user-defined **parameters** (input/output) can be defined as sequences in the component properties



dialog box.

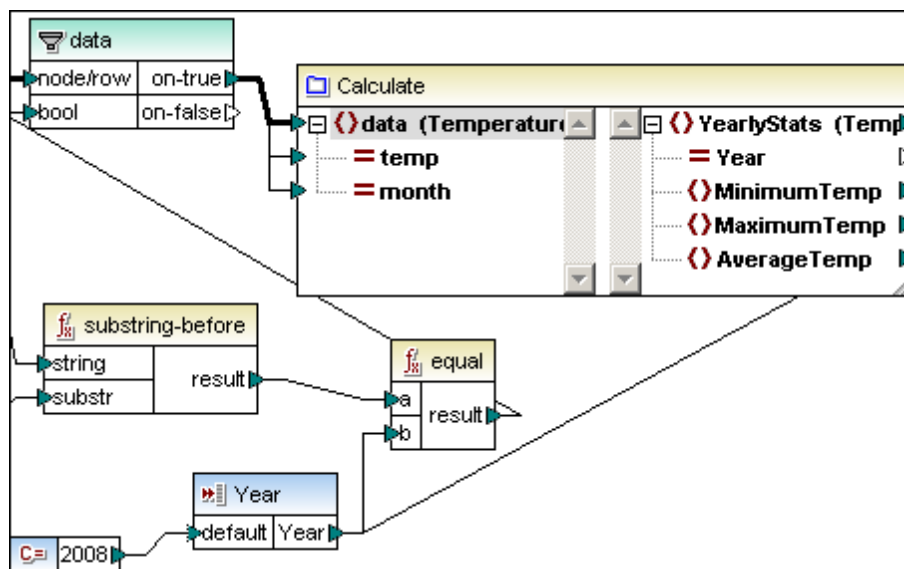
Aggregate functions, e.g. min, max, avg, etc., can use this type of input to supply a single specific value from the input sequence.

When the **"Input is a Sequence"** check box is active, the component handles the input as a sequence. When inactive, input is handled as a single value.



This type of input data, sequence or non-sequence, determines **how often** the function is called.

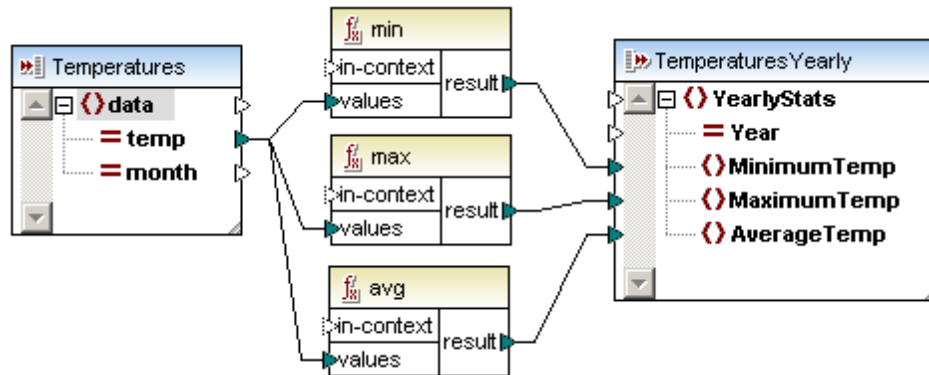
- When connected to a **sequence** parameter the user-defined function is called only **once** and the complete sequence is passed into the user-defined function.



The screenshot shows the user-defined function "Calculate" of the



"InputsSequence.mfd" mapping in the ...\\MapForceExamples folder. The **Temperatures** input component (shown below) is defined as a sequence.



- When connected to a **non-sequence** parameter, the user-defined function is called **once** for **each** single item in the sequence.

Please note:

The sequence setting of input/output parameters is ignored when the user-defined function is of type [inline](#).

Connecting an empty sequence to a non-sequence parameter has the result that the function is not called at all.

This can happen if the source structure has **optional** items, or when a filter condition returns no matching items. To avoid this, either use the [substitute-missing](#) function before the function input to ensure that the sequence is never empty, or set the parameter to sequence, and add handling for the empty sequence inside the function.

When a function passes a sequence of multiple values to its output component, and the output component is not set to sequence, only the first result is used when the function is called.

## 7.2.2 Inline and regular user-defined functions

Inline functions differ fundamentally from regular functions, in the way that they are implemented when code is generated.

- The code for **inline** type functions is inserted at **all locations** where the user-defined functions are called/used
- The code of a **regular** function is implemented as a function call.

Inline functions thus behave as if they had been replaced by their implementation. That makes them ideal for breaking down a complex mapping into smaller encapsulated parts.

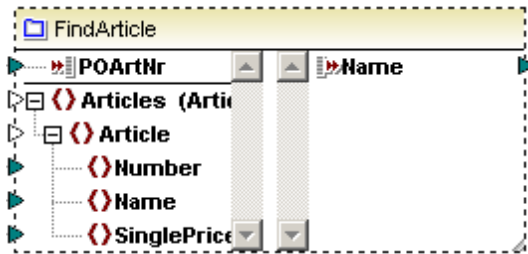
Please note:

using **inline** functions can significantly increase the amount of generated program code! The user-defined function code is actually inserted at all locations where the function is called/used, and thus increases the code size substantially - as opposed to using a



regular function.

**INLINE** user-defined functions are shown with a **dashed** outline:



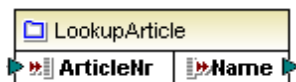
**Inline** user-defined functions **support**:

- **Multiple output components** within a function

**do not support**:

- The setting of a priority context on a parameter
- Recursive calls to an inline user-defined function

**REGULAR** user-defined functions i.e. non-inline functions are shown with a **solid** outline:



**Regular** (non-inline) user-defined functions **support**:

- Only a **single output component** within a function
- **Recursive** calls (where the exit condition must be supplied, e.g. use an If-Else condition where one branch, or value, exits the recursion)
- Setting a priority context on a parameter

Please note:

Although regular functions **do not** support multiple output components, they **can be created** in this type of function. However, an error message appears when you try to generate code, or preview the result of the mapping.

If you are not using recursion in your function, you can change the type of the function to "inline".

**do not support**:

- Direct connection of filters to simple non-sequence **input** components
- Sequence or aggregate functions on simple input components (like exists, substitute-missing, sum, group-by, ...)

### Code generation

The implementation of a regular user-defined function is generated only once as a callable XSLT template or function. Each user-defined function component generates code for a **function call**, where inputs are passed as parameters, and the output is the function (component) return value.



At runtime, all the input parameter values are evaluated first, then the function is called for each occurrence of the input data. See [Function parameters](#) for details about this process.

#### To change the user-defined function "type":

1. Double click the user-defined function to see its constituent components.
2. Select the menu option **Function | Function settings** and click the "Inlined use" check box.

#### User-defined functions and Copy-all connections

When creating Copy-all connections between a schema and a complex user-defined function parameter, the two components must be based on the same schema! It is not necessary that they both have the same root elements however. Please see "[Complex output components - defining](#)" for an example.

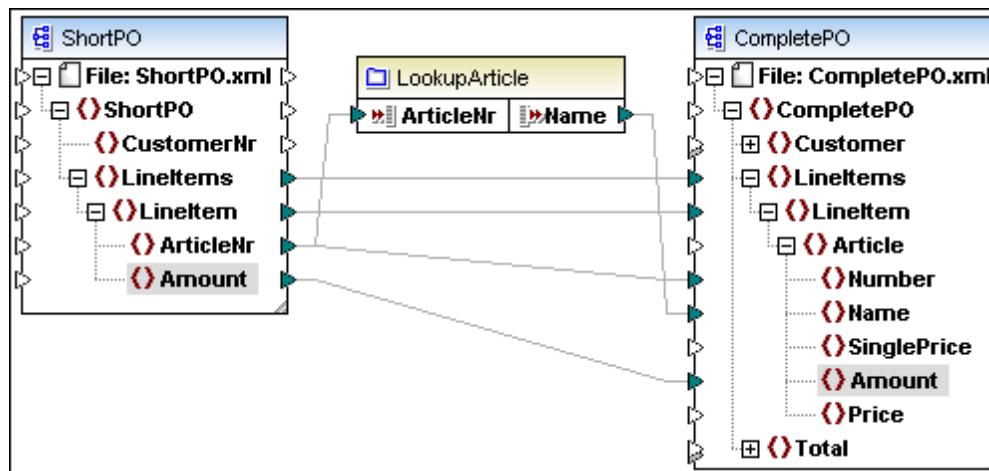
### 7.2.3 Creating a simple look-up function

This example is provided as the **lookup-standard.mfd** file available in the [... \MapForceExamples](#) folder.

Aim:

To create a generic look-up function that:

- supplies Articles/Number data from the Articles XML file, to be compared to Article numbers of a different XML file, ShortPO in this case.

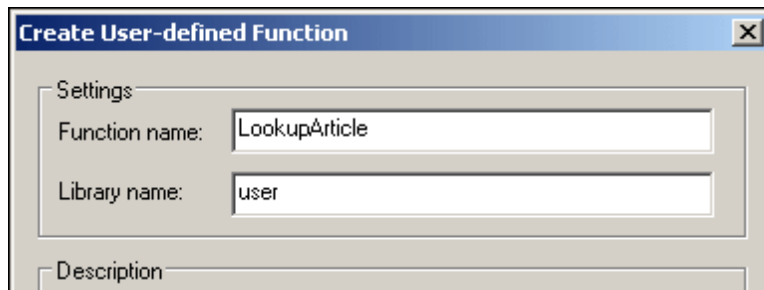


- Insert the ShortPO.xsd and assign ShortPO.xml as the source XML file.
- Insert the CompletePO.xsd schema file, and select CompletePO as the root element.
- Insert a new user-defined function using the method described below.

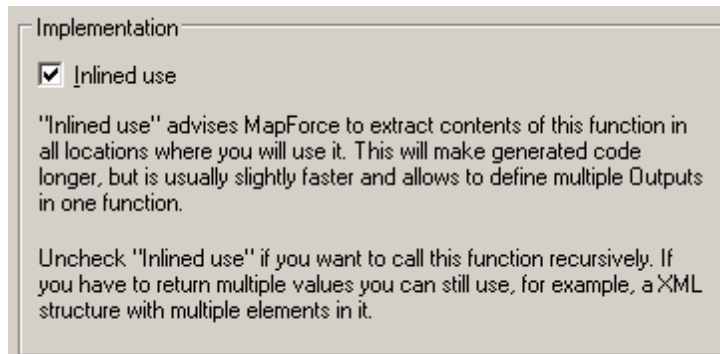
#### To create a user-defined function:

1. Select the menu option **Function | Create User-defined function**.
2. Enter the name of the function e.g. LookupArticle.

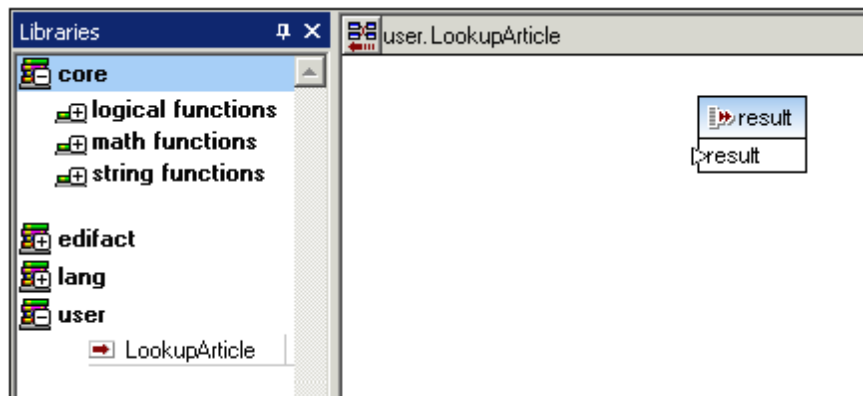




3. Uncheck the "Inlined use" check box and click OK to confirm





A tab only containing only one item, an output function currently called "result", is displayed.

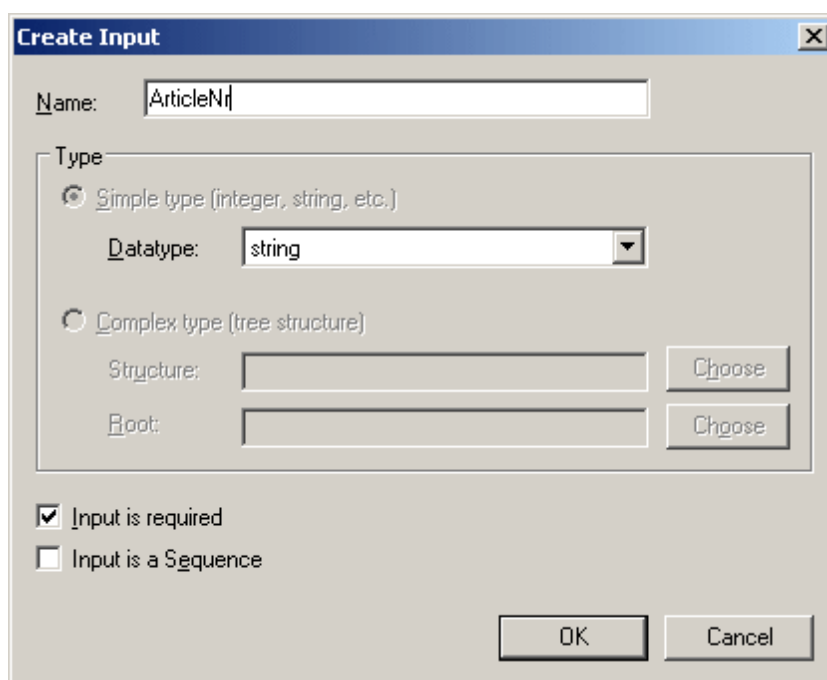


This is the working area used to define the user-defined function.

A new library has been created in the Libraries pane with the name "user" and the function name "LookupArticle".

3. Click the **Insert Schema/XML file** icon  to insert the **Articles** schema and select the XML file of the same name to act as the data source.
4. Click the **Insert input component** icon  to insert an input component.
5. Enter the name of the input parameter, ArticleNr in this case, and click OK.





**Create Input**

Name:

Type

☒ Simple type (integer, string, etc.)

Datatype:

☐ Complex type (tree structure)


Structure:

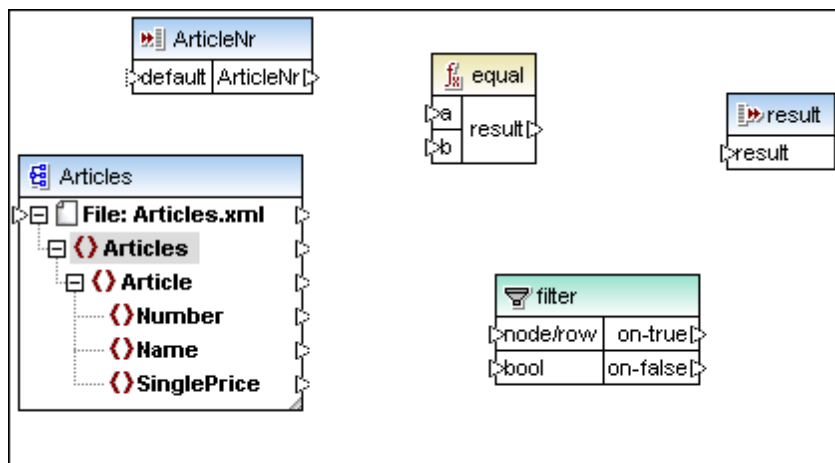
Root:

☒ Input is required

☐ Input is a Sequence

This component acts as a data input to the user-defined function and supplies the input icon of the user-defined function.

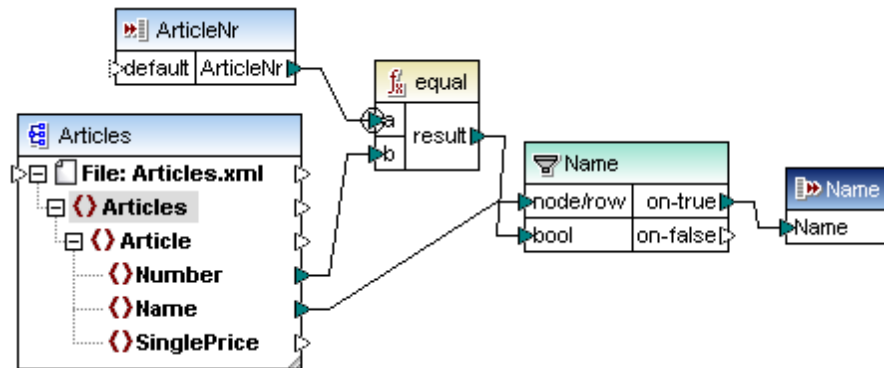
6. Insert an **"equal"** component by dragging it from the core library/logical functions group.
7. Insert a **filter** component by clicking the Insert Filter icon  in the toolbar.



Use the diagram below as an aid to creating the mappings in the user-defined function, please take note of the following:

8. Right click the **a** parameter and select **Priority context** from the pop up menu.
9. **Double click** the output function and enter the name of the output parameter, in this case **"Name"**.





This ends the definition of the user-defined function.

Please note:

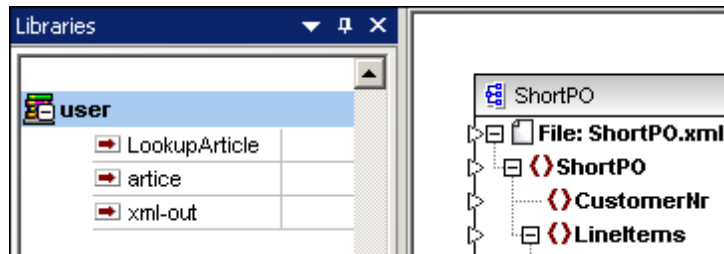
Double clicking the **input** and **output** functions opens a dialog box in which you can change the name and datatype of the input parameter, as well as define if the function is to have an input icon (Input is required) and additionally if it should be defined as a sequence.

This user-defined function:

- has one **input** function, ArticleNr, which will receive data from the ShortPO XML file.
- **compares** the ShortPO ArticleNr, with the Article/Number from the **Articles** XML instance file, inserted into the user-defined function for this purpose.
- uses a **filter** component to forward the Article/Name records to the output component, if the comparison returns true.
- has one output function, Name, which will forward the Article Name records to the CompletePO XML file.

10. Click the Home icon  to return to the main mapping.

The LookupArticle user-defined function, is now available under the **user** library.

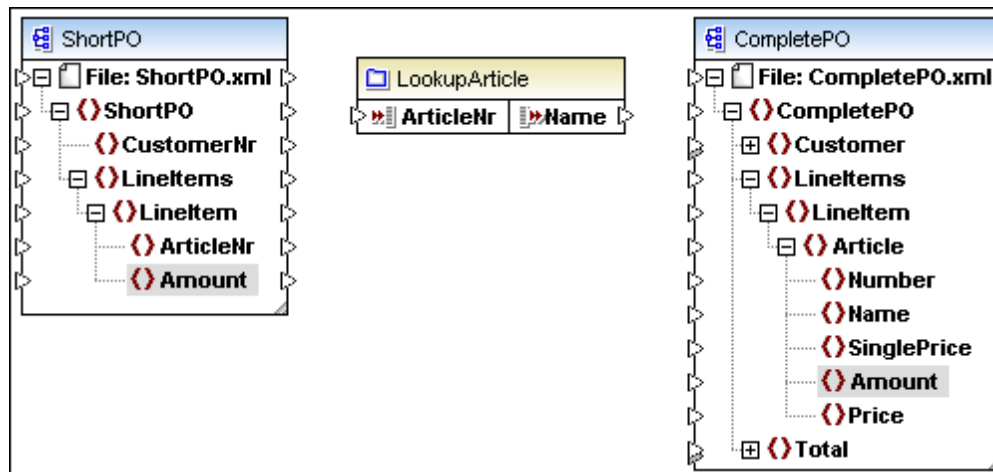


11. Drag the LookupArticle function into the Mapping window.

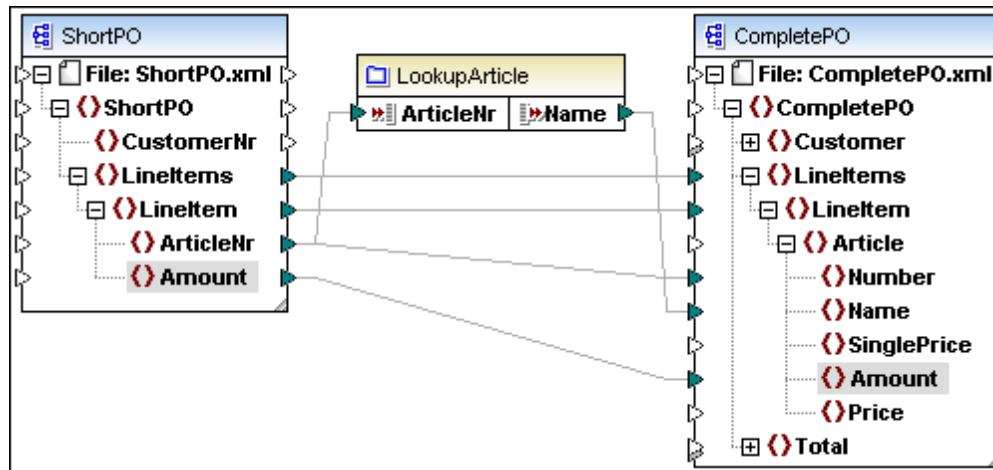
The user-defined function is displayed:

- with its name "LookupArticle" in the title/function bar,
- with named input and output icons.





10. Create the connections displayed in the graphic below and click the Output tab to see the result of the mapping.

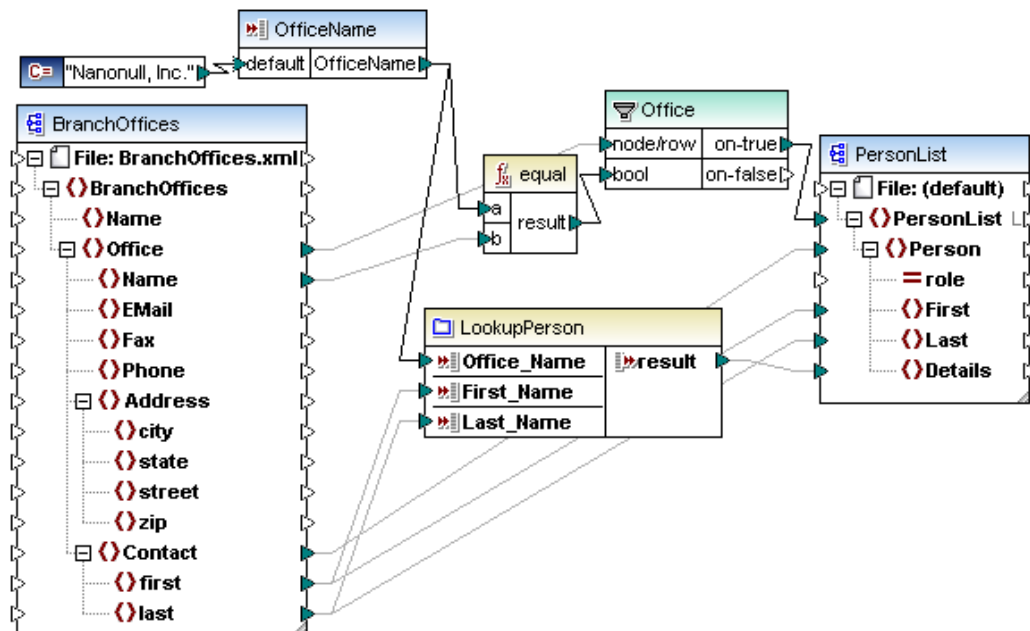


## 7.2.4 User-defined function - example

The **PersonListByBranchOffice.mfd** file available in the <Documents>\Altova\MapForce2018\MapForceExamples\ folder illustrates the following features:

- Nested User-defined functions e.g. **LookupPerson**
- Look-up functions that generate a string output e.g. **LookupPerson**
- **Optional** input-parameters which can also supply a **default** value e.g. the **EqualAnd** component (contained in the **LookupPerson** component)
- **Configurable** input parameters, which can also double as a command line parameter(s) when executing the generated mapping code!

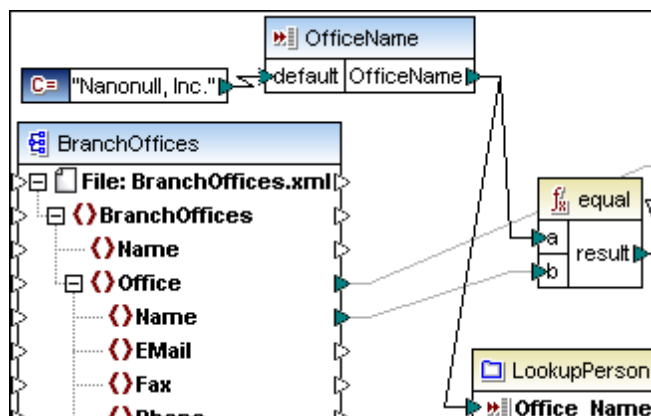




### Configurable input parameters

The input component (OfficeName) receives data supplied when a mapping is executed. This is possible in two ways:

- as a **command line** parameter when executing the generated code, e.g. Mapping.exe / OfficeName "Nanonull Partners, Inc."
- as a **preview** value when using the Built-in execution engine to preview the data in the Output window.



### To define the Input value:

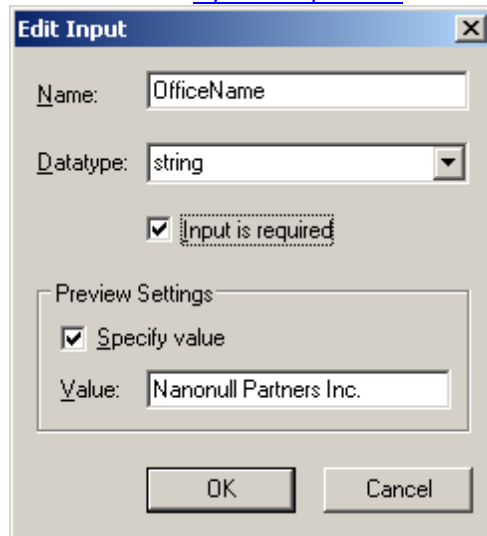
1. Double click the input component and enter a different value in the "Value" text box of the Preview Mode group e.g. "Nanonull Partners, Inc.", and click OK to confirm.
2. Click the Output tab to see the effect.



A different set of persons are now displayed.

Please note that the data entered in this dialog box is only used in "**preview**" mode i.e. when clicking the Output tab. If a value is not entered, or the check box is deactivated, then the data mapped to the input icon "default" is used.

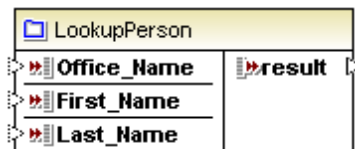
Please see [Input Components](#) for more information.



The "Edit Input" dialog box contains the following fields and controls:

- Name:** A text field containing "OfficeName".
- Datatype:** A dropdown menu set to "string".
- Input is required:** A checked checkbox.
- Preview Settings:** A section containing:
  - Specify value:** A checked checkbox.
  - Value:** A text field containing "Nanonull Partners Inc."
- Buttons:** "OK" and "Cancel" buttons at the bottom.

### LookupPerson component

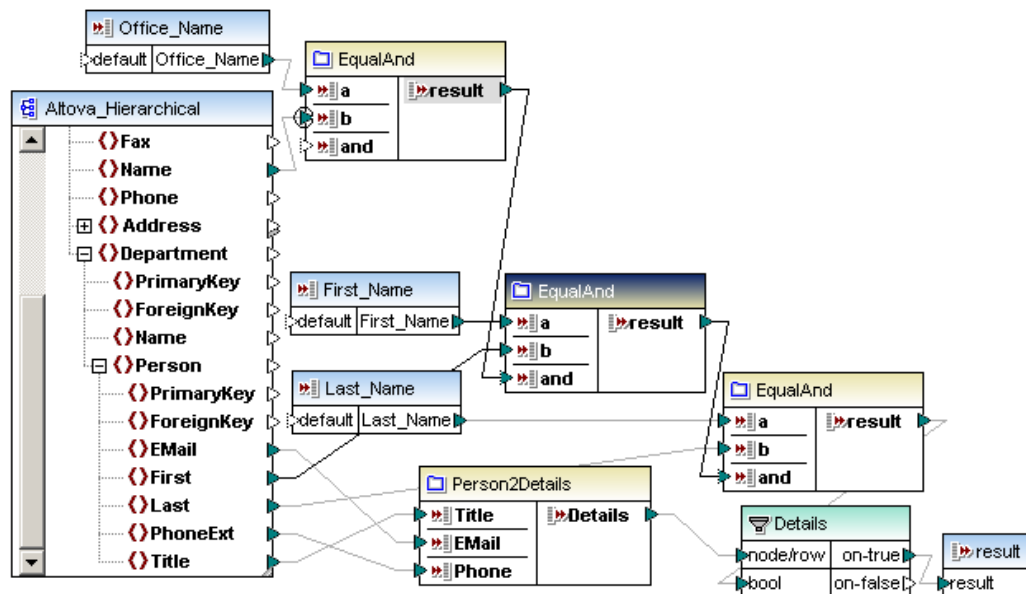


Double clicking this user-defined component displays its constituent components shown below. What this component does is:

- **Compares** the Office, First, and Last names of BranchOffices.xml, with the same fields of the Altaova\_Hierarchical.xml file, using the **input** components and the **EqualAnd** user-defined components.
- **Combines** the Email, PhoneExt and Title items using the **Person2Details** user-defined function
- **Passes on** the combined person data to the **output** component if the previous EqualAnd comparisons are all true (i.e. supplied "true" to the filter component).

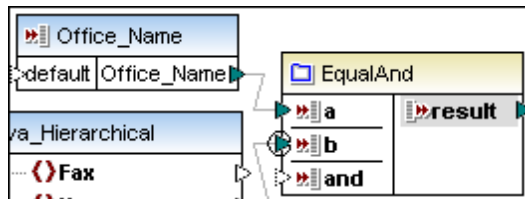
A user-defined function always outputs a value, which may even be an empty string! This would be the case if the filter component bool value is false. Only an empty string would be output instead of data supplied by the Person2Details component.





- The three **input** components, Office\_Name, First\_Name, Last\_Name, receive their data from the BranchOffices.xml file.
- The **EqualAnd** component compares two values and provides an **optional** comparison value, as well as a default value.
- Person2Details combines three person fields and passes on the result to the filter component.

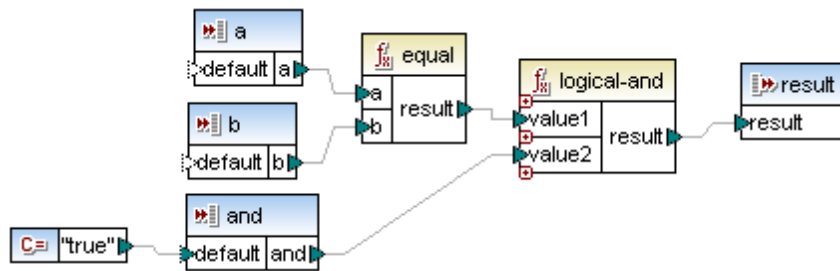
### EqualAnd component



Double clicking this user-defined component displays its constituent components shown below. What this component does is:

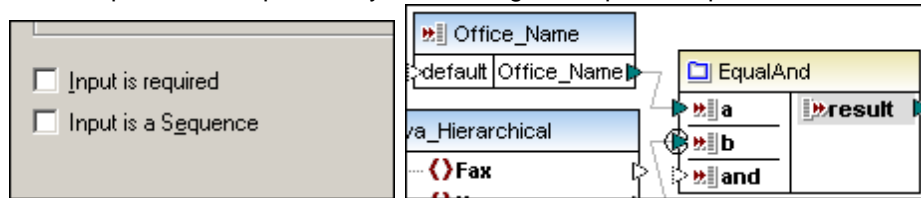
- Compare two input parameters **a** and **b**, and pass the result on to the logical-and component. Note that the **b** parameter has been defined as the **priority context** (right click the icon to do so). This ensures that the person data of the specific office, supplied by the input parameter **a**, is processed first.
- **Logical-and** the result of the first comparison, with an **optional** input parameter, "and".
- Pass on the boolean value of this comparison to the output parameter.





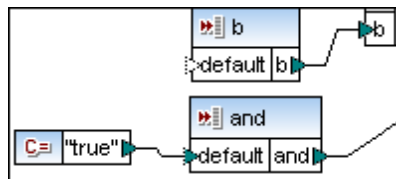
### Optional parameters

Double clicking the "and" parameter, of the EqualAnd user-defined function shown above, allows you to make parameters optional, by unchecking the "Input is required" check box.



If "Input is required" is **unchecked**, then:

- A mapping connector is not required for the input icon of this user-defined function, e.g. the **and** parameter of the first EqualAnd function, does not have an input connector. The input icon has a dashed outline to show this visually.
- A **default** value can be supplied by connecting a component, within the user-defined function e.g. using a constant component containing the value "true".



- A mapping from another item, mapped to the optional Input, takes precedence over the default value. E.g. the "and" parameter of second EqualAnd function, receives input data from the "result" parameter of the first EqualAnd user-defined function.

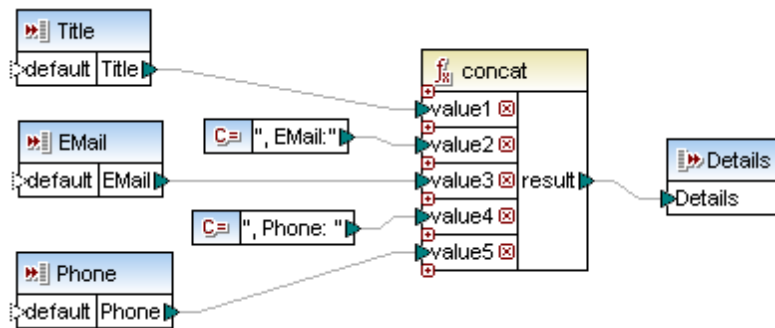
### Person2Details component



Double clicking this user-defined component displays its constituent components shown below. What this component does is:

- Concatenate three inputs and pass on the result string to the output parameter.
- Double clicking an output parameter allows you to change the parameter name (Details), and select the datatype (String).





### 7.2.5 Complex user-defined function - XML node as input

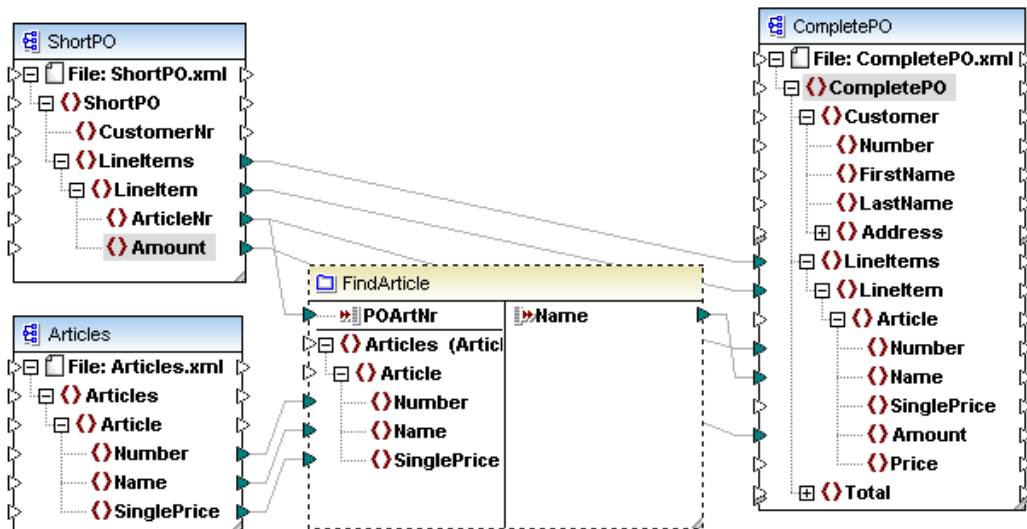
This example is provided as the **lookup-udf-in.mfd** file available in the [...\MapForceExamples](#) folder. This section illustrates how to define an inline user-defined function that contains a complex input component.

Note that the user-defined function "FindArticle" consists of two halves.

The left half contains the input parameters:

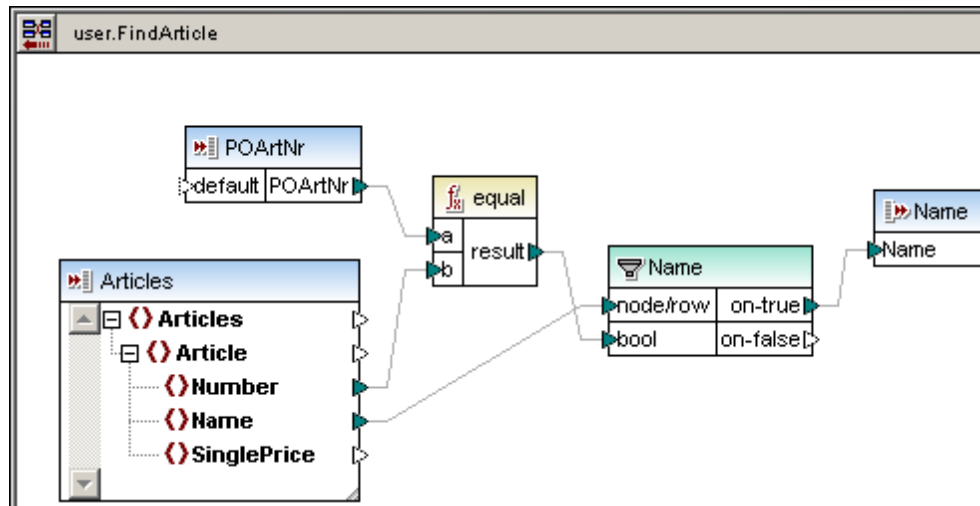
- a simple input parameter **POArtNr**
- a complex input component **Articles**, with mappings directly to its XML child nodes

The right half contains a simple output parameter called "**Name**".



The screenshot below shows the constituent components of the user-defined function, the two input components to the left and the output component to the right.

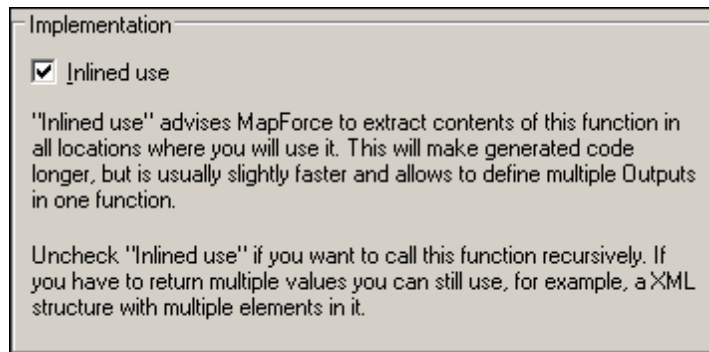





### 7.2.5.1 Defining Complex Input Components

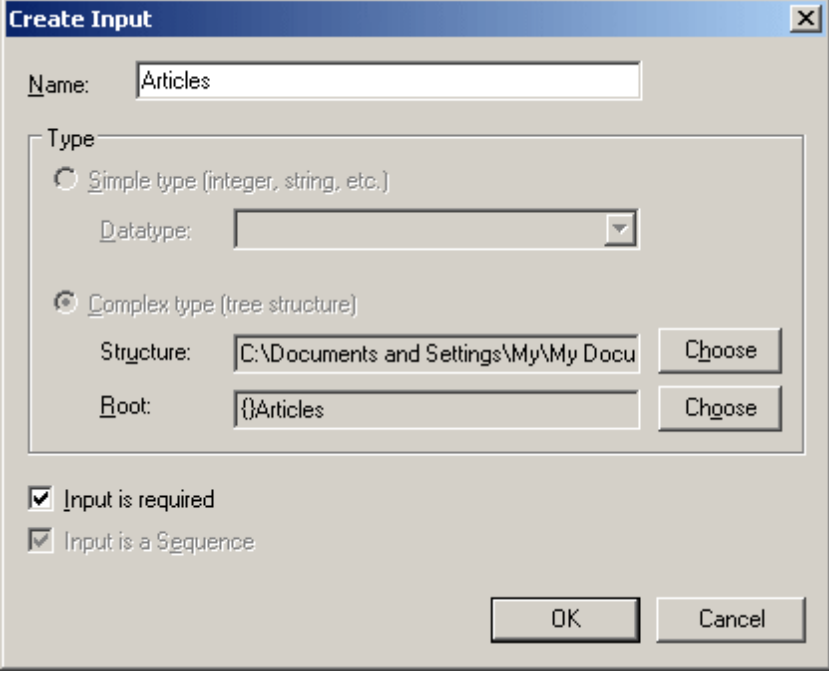
Follow these steps to create a function that takes an XML structure as input parameter:

1. Create a user-defined function in the usual manner, i.e. **Function | Create User-Defined function** and click OK to confirm. Note that the **Inlined use** check box is automatically selected.



2. Click the **Insert input component** icon  in the icon bar.
3. Enter the name of the input component into the Name field.



The image shows a 'Create Input' dialog box with a title bar and a close button. It contains a 'Name' field with the text 'Articles'. Below this is a 'Type' section with two radio buttons: 'Simple type (integer, string, etc.)' and 'Complex type (tree structure)'. The 'Complex type' radio button is selected. Under 'Simple type' is a 'Datatype' dropdown menu. Under 'Complex type' are two fields: 'Structure' with the text 'C:\Documents and Settings\My\My Docu' and a 'Choose' button, and 'Root' with the text '{}Articles' and a 'Choose' button. At the bottom are two checked checkboxes: 'Input is required' and 'Input is a Sequence'. At the very bottom are 'OK' and 'Cancel' buttons.

**Create Input**

Name:

Type

☐ Simple type (integer, string, etc.)

Datatype:

☒ Complex type (tree structure)

Structure:

Root:

☒ Input is required

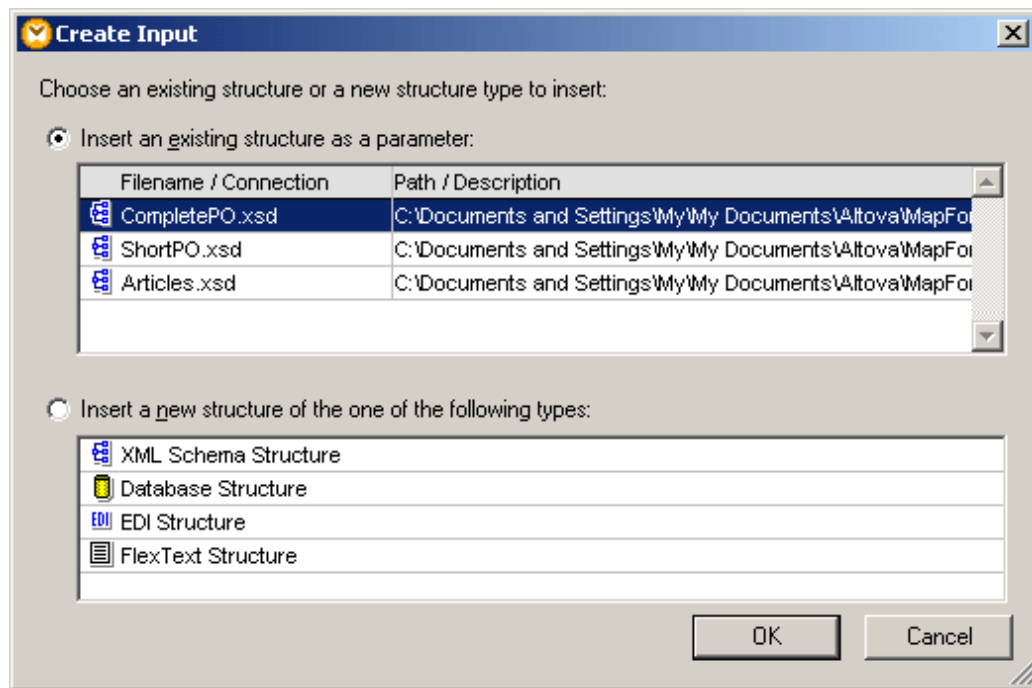
☒ Input is a Sequence

4. Click the **Complex type (tree structure)** radio button, then click the "**Choose**" button next to the Structure field. This opens another dialog box.

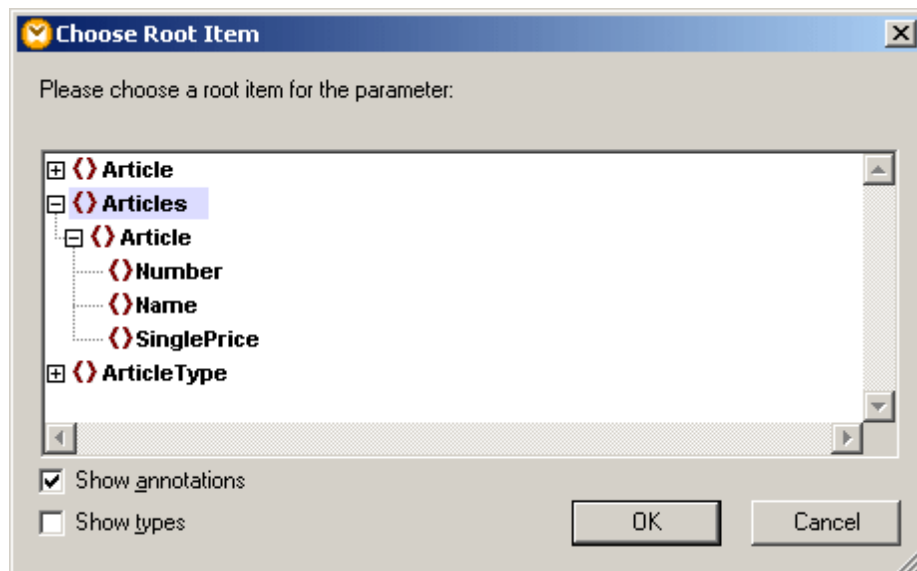
The top list box displays the **existing** components in the mapping (three schemas if you opened the example mapping). Note that this list contains all of the components that have been inserted into the active mapping: e.g. XML schema file.


The lower list box allows you to select a new complex data structure i.e. XML Schema file.





5. Click "Insert a new structure..." radio button, select the XML Schema Structure entry, and click OK to continue.
6. Select **Articles.xsd** from the "Open" dialog box.
7. Click the element that you would like to become the root element in the component, e.g. Articles, and click OK, then OK again to close both dialog boxes.

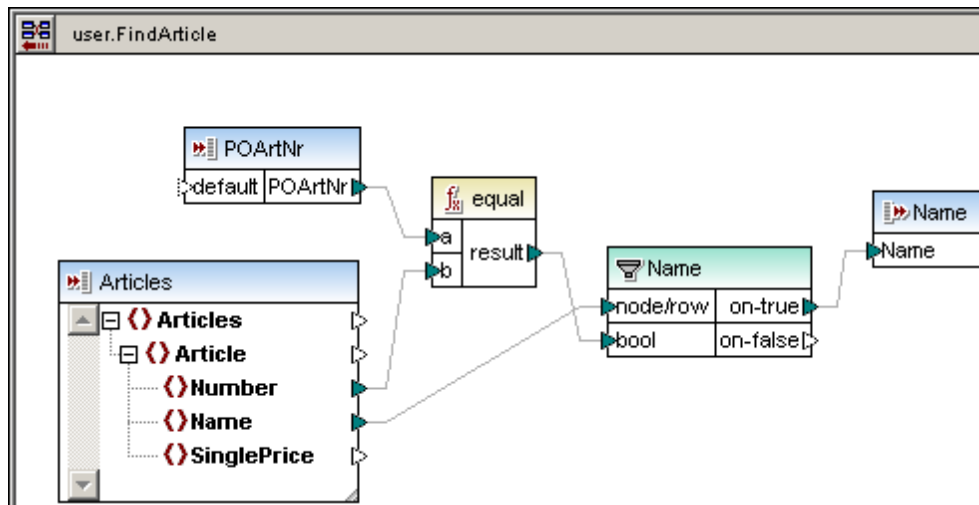



The Articles component is inserted into the user-defined function. Please note the input icon  to the left of the component name. This shows that the component is used as a complex input component.





8. Insert the rest of the components as shown in the screenshot below, namely: a second "simple" input component (called POArtNr), filter, equal and output component (called Name), and connect them as shown.

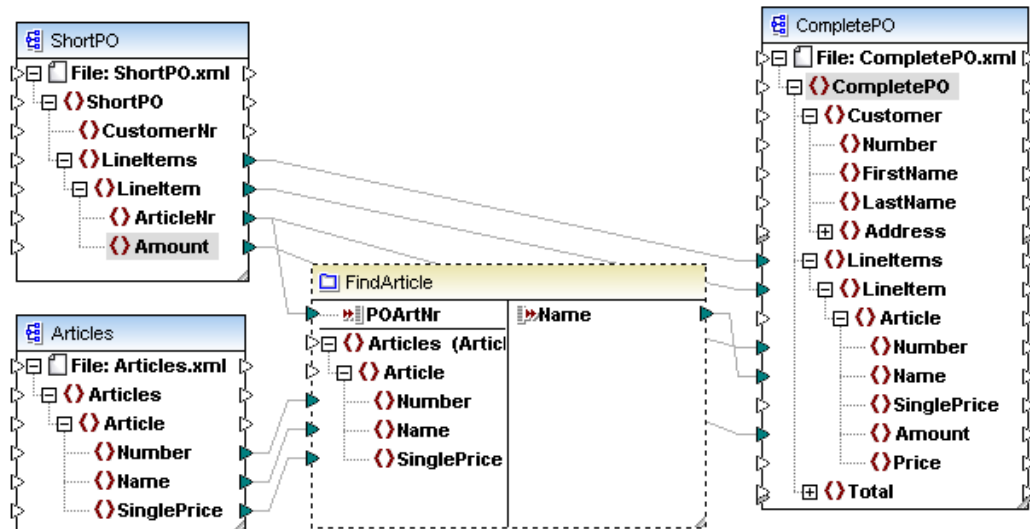


- The **Articles** input component receives its data from **outside** of the user-defined function. Input icons that allow mapping to this component, are available there.
  - An XML **instance** file to provide data from within the user-defined function, cannot be assigned to a complex input component.
  - The other input component POArtNr, supplies the ShortPO article number data to which the **Article | Number** is compared.
  - The filter component filters the records where both numbers are identical, and passes them on to the output component.
10. Click the Home icon  to return to the mapping.
11. Drag the newly created user-defined component from the Libraries pane into the mapping.



12. Create the connections as shown in the screenshot below.





The left half contains the input parameters to which items from two schema/xml files are mapped:

- **ShortPO** supplies the data for the input component **POArtNr**.
- **Articles** supplies the data for the complex input component. The Articles.xml instance file was assigned to the Articles schema file when the component was inserted.
- The complex input component **Articles** with its XML child nodes, to which data has been mapped from the Articles component.

The right half contains a simple output parameter called "**Name**", which passes the filtered line items which have the same Article number to the "Name" item of **CompletePO**.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <CompletePO xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3  <LinItems>
4  <LinItem>
5  <Article>
6  <Number>3</Number>
7  <Name>Pants</Name>
8  <Amount>5</Amount>
9  </Article>
10 </LinItem>
11 <LinItem>
12 <Article>
13 <Number>1</Number>
14 <Name>T-Shirt</Name>
15 <Amount>17</Amount>
16 </Article>
17 </LinItem>
18 </LinItems>
19 </CompletePO>

```

**Note:** When creating **Copy-all** connections between a schema and a user-defined function parameter, the two components must be based on the same schema. It is not necessary that they both have the same root elements however.



## 7.2.6 Complex user-defined function - XML node as output

This example is provided as the **lookup-udf-out.mfd** file available in the [...MapForceExamples](#) folder. What this section will show is how to define an inline user-defined function that allows a complex output component.

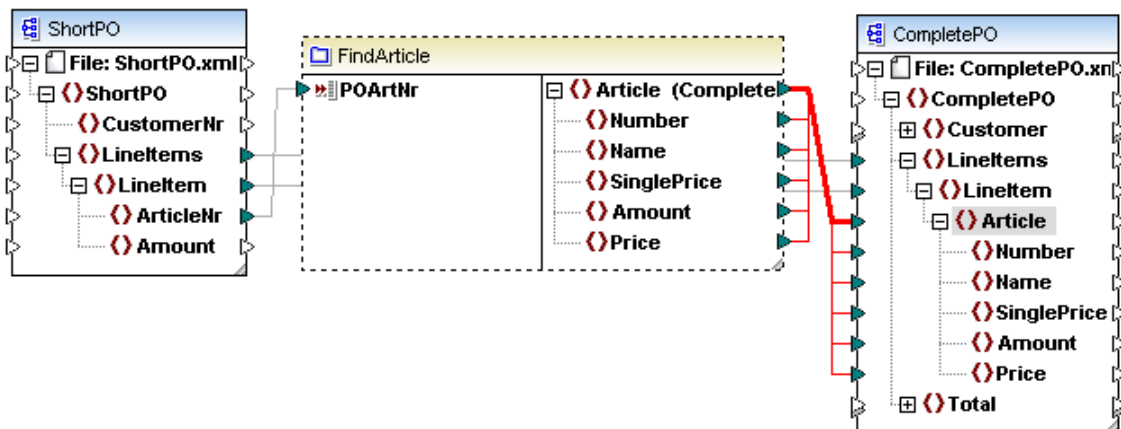
Note that the user-defined function FindArticle consists of two halves.

A left half which contains the input parameter:

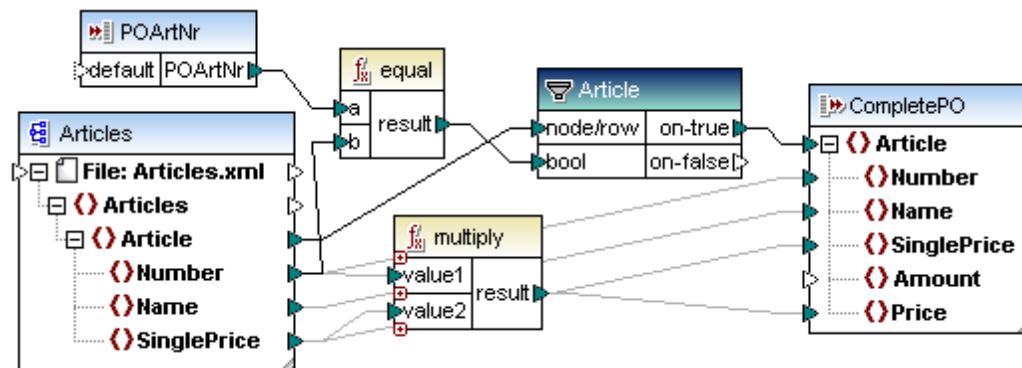
- a simple input parameter **POArtNr**

A right half which contains:

- a complex output component **Article (CompletePO)** with its XML child nodes mapped to CompletePO.



The screenshot below shows the constituent components of the user-defined function, the input components at left and the complex output component at right.

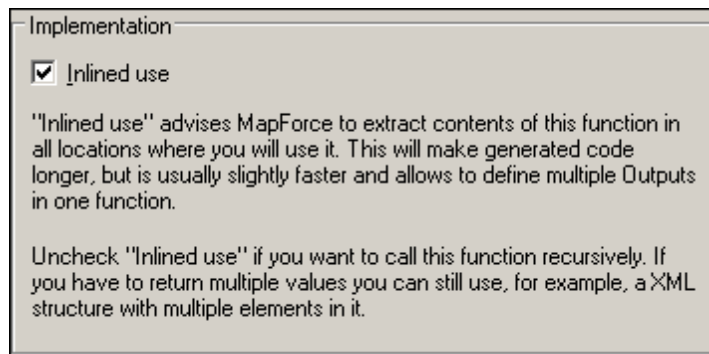



### 7.2.6.1 Defining Complex Output Components

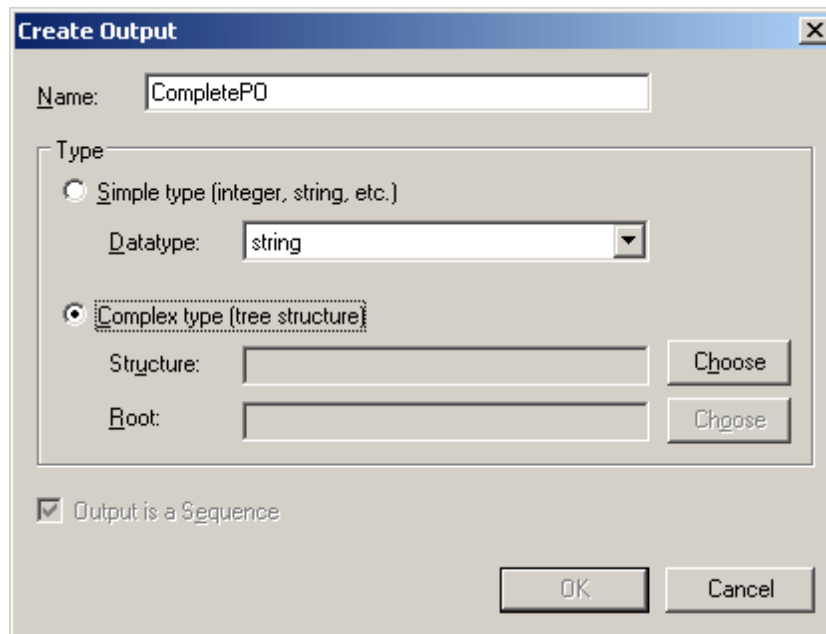
Follow these steps to create a function that returns an XML structure as output parameter:

1. Create a user-defined function in the usual manner, i.e. **Function | Create User-Defined function** name it **FindArticle**, and click OK to confirm. Note that the **Inline...** option is automatically selected.





2. Click the Insert **Output** icon  in the icon bar, and enter a name e.g. CompletePO.

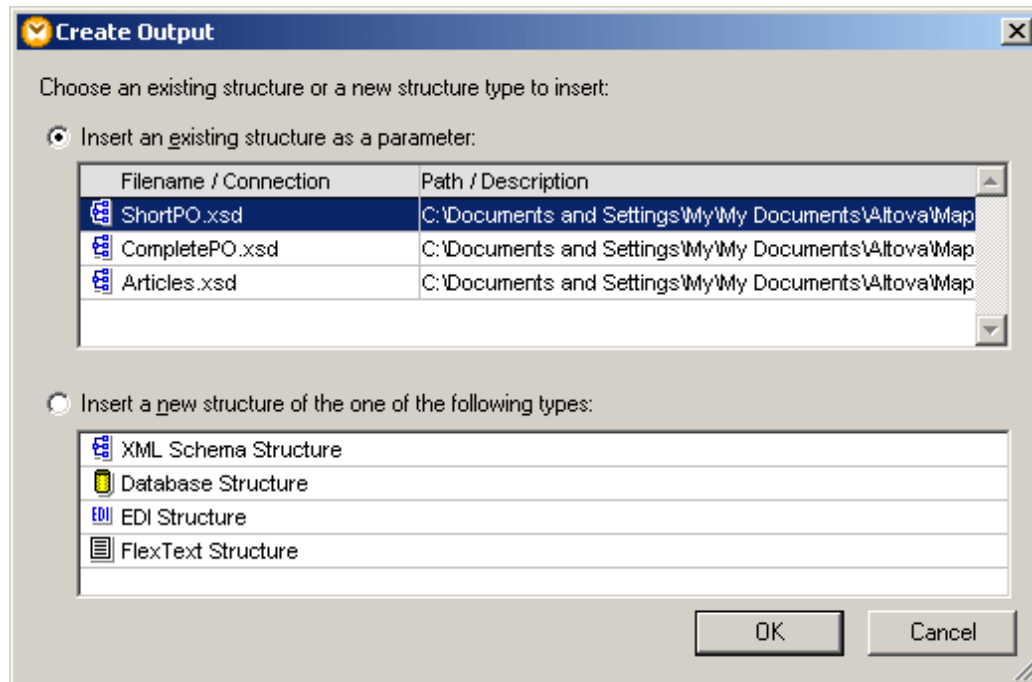


3. Click the **Complex type...** radio button, then click the "**Choose**" button. This opens another dialog box.

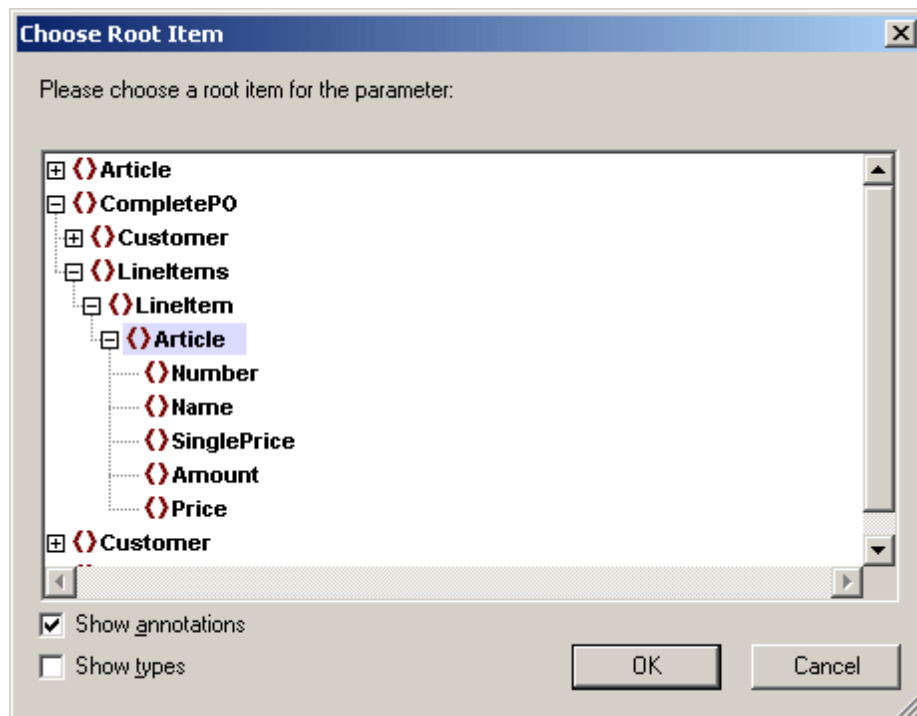
The top list box displays the **existing** components in the mapping, (three schemas if you opened the example file). Note that this list contains all of the components that have been inserted into the active mapping: e.g. XML Schema file.

The lower list box allows you to select a new complex data structure i.e. XML Schema file.






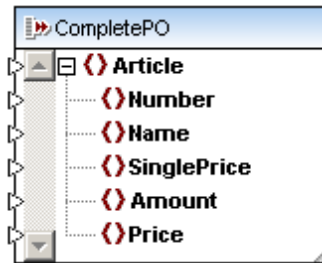
4. Click "Insert new structure..." radio button, select the XML Schema Structure entry, and click OK to continue.
5. Select the **CompletePO.xsd** from the "Open" dialog box.
6. Click the element that you would like to become the root element in the component, e.g. **Article**, and click OK, then OK again to close the dialog boxes.



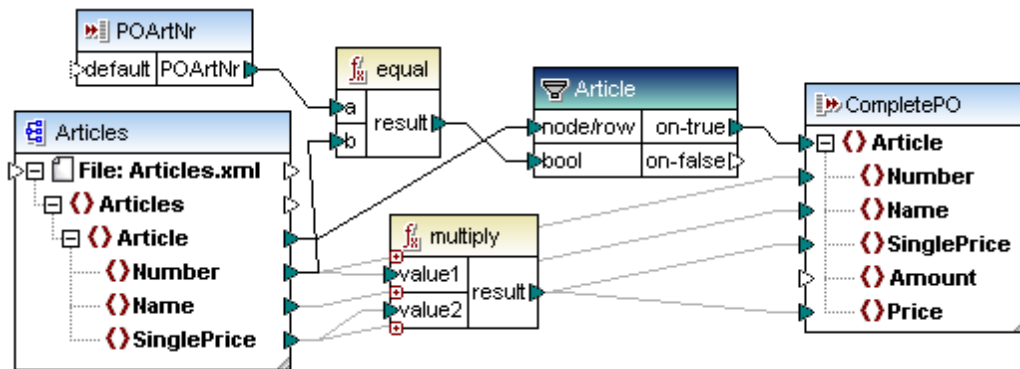
The CompletePO component is inserted into the user-defined function. Please note the




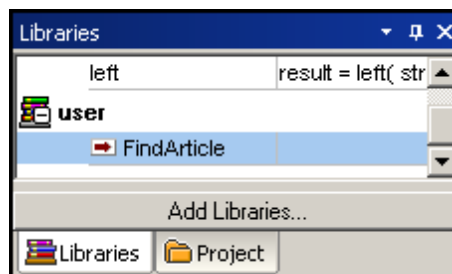
output icon  to the left of the component name. This shows that the component is used as a complex output component.



7. Insert the Articles schema/XML file into the user-defined function and assign the **Articles.xml** as the XML instance.
8. Insert the rest of the components as shown in the screenshot below, namely: the "simple" input components (POArtNr), filter, equal and multiply components, and connect them as shown.



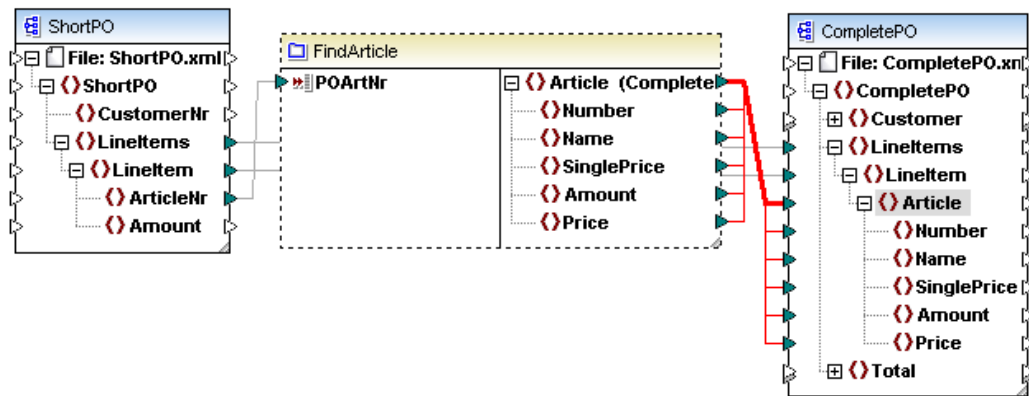
- The **Articles** component receives its data from the Articles.xml instance file, within the user-defined function.
  - The input components supply the POArtNr (article number) and Amount data to which the Articles | Number & Price are compared.
  - The filter component filters the records where both numbers are identical, and passes them on to the CompletePO output component.
9. Click the Home icon  to return to the mapping.
  10. Drag the newly created user-defined component from the Libraries pane into the mapping.



11. Create the connections as shown in the screenshot below.  
Having created the Article (CompletePO) connector to the target, right click it and select



"Copy-all" from the context menu. The rest of the connectors are automatically generated, and are highlighted in the screenshot below.

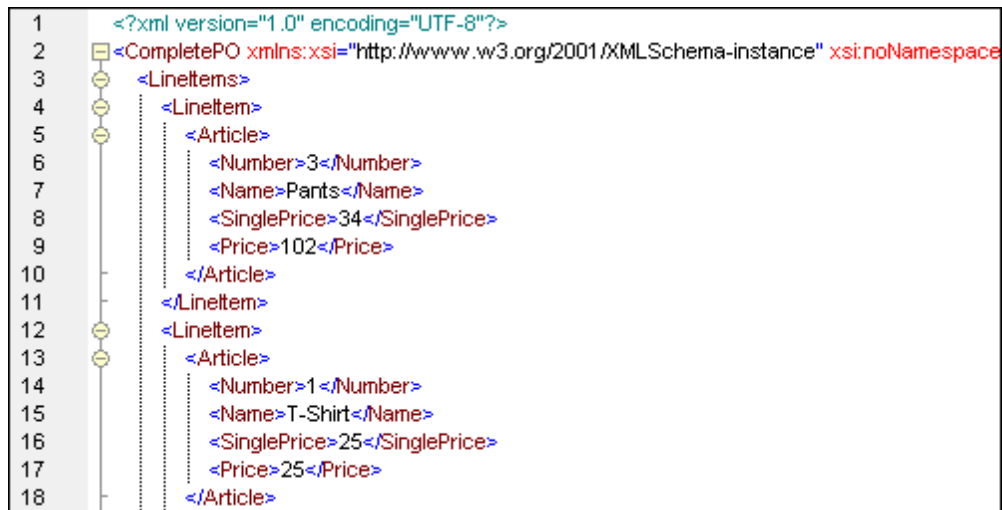


Please note:

When creating **Copy-all** connections between a schema and a user-defined function of type "Inline", the two components must be based on the same schema. It is not necessary that they both have the same root elements however.

The left half contains the input parameter to which a single item is mapped; ShortPO supplies the article number to the **POArtNr** input component.

The right half contains a complex output component called "**Article (CompletePO)**" with its XML child nodes, which maps the filtered items, of the same Article number, to CompletePO.



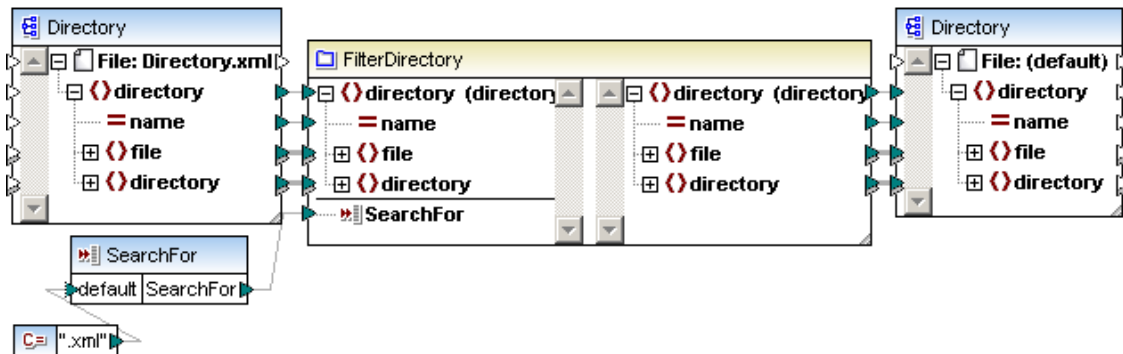
## 7.2.7 Recursive user-defined mapping

This section will describe how the mapping **RecursiveDirectoryFilter.mfd**, available in the ... \MapForceExamples folder, was created and how recursive mappings are designed. The MapForceExamples project folder contains further examples of recursive mappings.

The screenshot below shows the finished mapping containing the recursive user-defined function



FilterDirectory, the aim being to filter a list of the .xml files in the source file.



The **source file** that contains the file and directory data for this mapping is Directory.xml. This XML file supplies the directory and file data in the hierarchical form you see below.

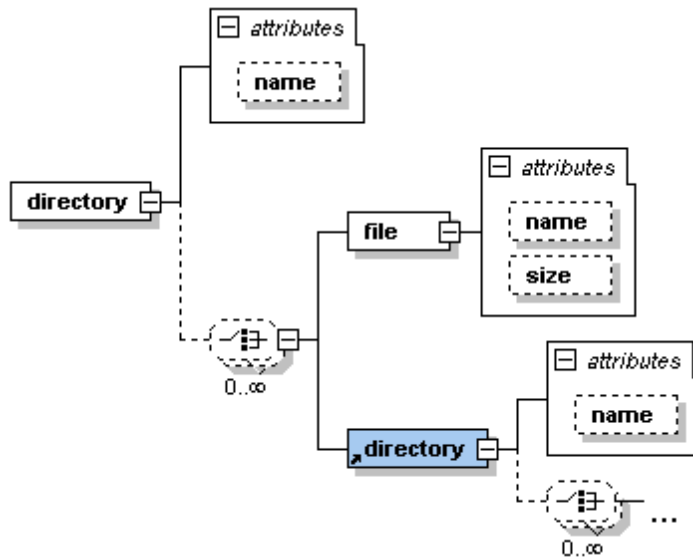
```

24      <directory name="output">
25          <file name="examplesite1.css" size="3174"/>
26      <directory name="images">
27          <file name="blank.gif" size="88"/>
28          <file name="block_file.gif" size="13179"/>
29          <file name="block_schema.gif" size="9211"/>
30          <file name="nav_file.gif" size="60868"/>
31          <file name="nav_schema.gif" size="6002"/>
32      </directory>
33  </directory>
34 </directory>
35 <directory name="Import">
36     <file name="altova.mdb" size="266240"/>
37     <file name="Data_shape.mdb" size="225280"/>
38 </directory>
39 <directory name="IndustryStandards">
40     <directory name="News">
41         <file name="high-tide.jpg" size="10793"/>
42         <file name="Newsml-example.xml" size="5004"/>
43         <file name="nitf-example.xml" size="9327"/>
44     </directory>

```

The XML schema file referenced by Directory.xml has a **recursive** element called "directory" which allows for any number of subdirectories and files below the directory element.

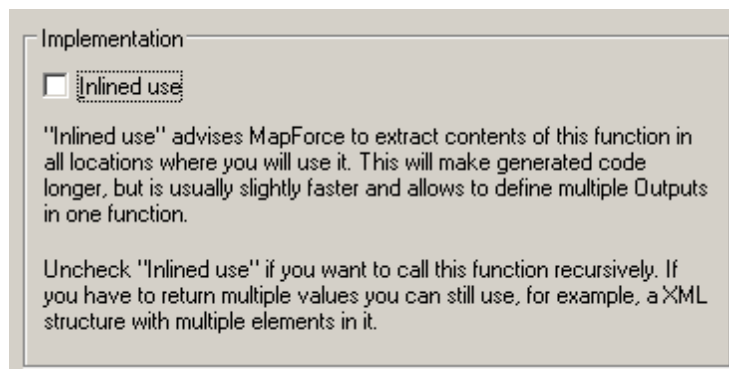




### 7.2.7.1 Defining a recursive user-defined function

Follow these steps to create a recursive user-defined function:

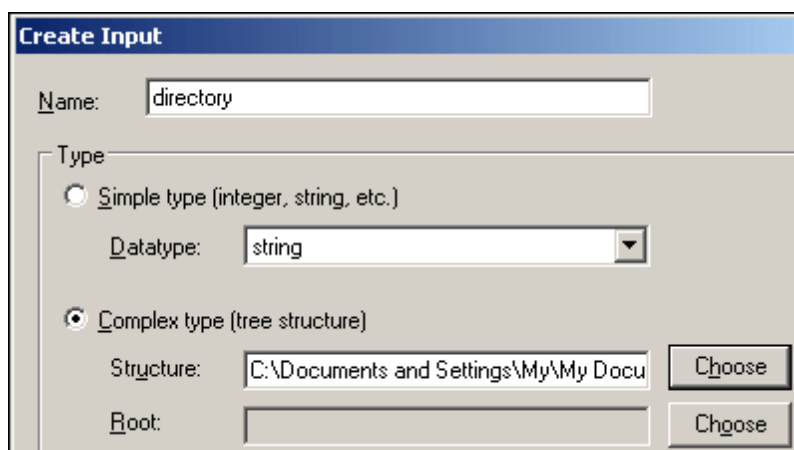
1. Select **Function | Create User defined Function** to start designing the function and enter a name e.g. FilterDirectory.
2. Make sure that you **deselect** the **Inlined Use** check box in the Implementation group, to make the function recursive, then click OK.



You are now in the **FilterDirectory** window where you create the user-defined function.

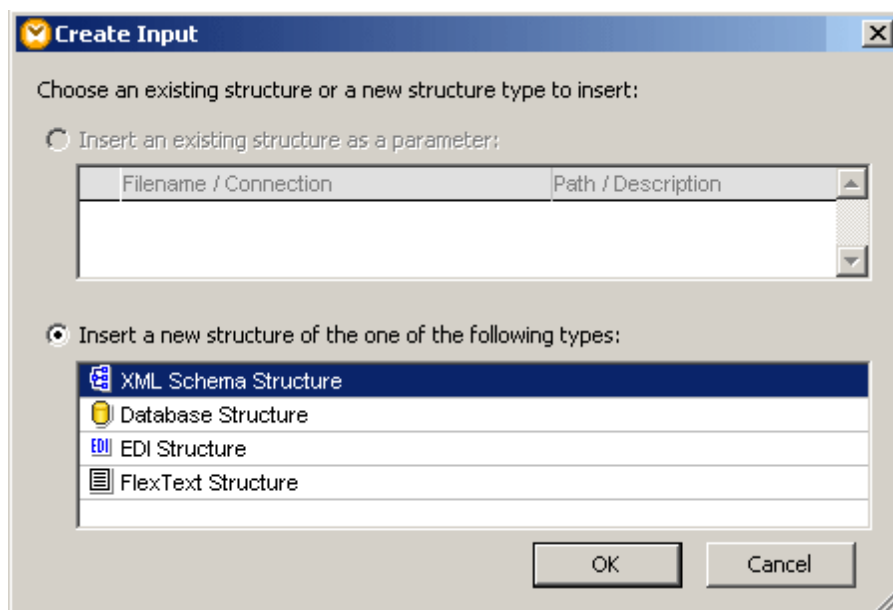
3. Select **Function | Insert Input** to insert an input component.
4. Give the component a name e.g. "directory" and click on the **Complex Type** (tree structure) radio button.





The "Create Input" dialog box is shown. The "Name" field contains "directory". Under the "Type" section, the "Simple type (integer, string, etc.)" radio button is selected, and the "Datatype" dropdown is set to "string". The "Complex type (tree structure)" radio button is also visible. Below it, the "Structure" field contains a file path, and the "Root" field is empty. Both have "Choose" buttons next to them.

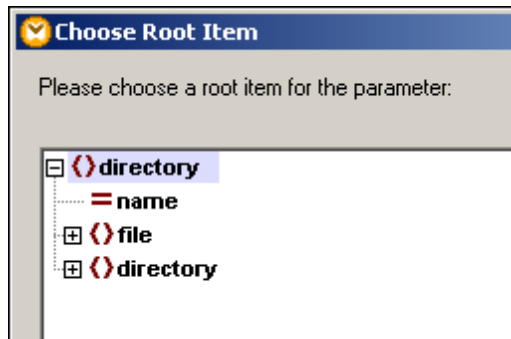
5. Click the **Choose** button, click the "XML Schema Structure" entry in the lower pane, then click OK.



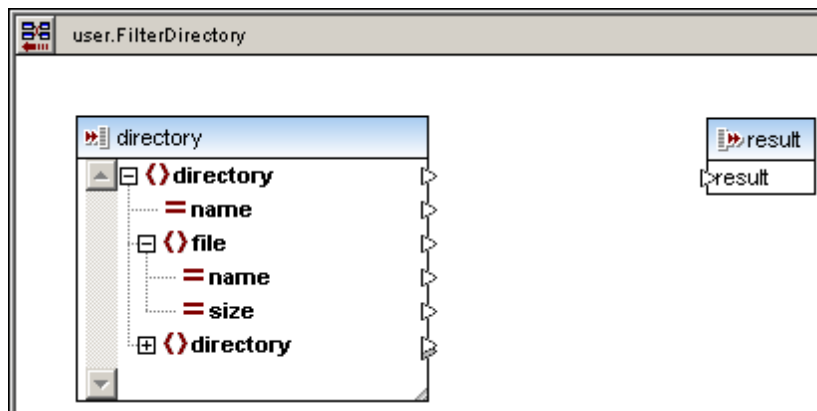
This is a second "Create Input" dialog box. It prompts the user to "Choose an existing structure or a new structure type to insert:". The "Insert an existing structure as a parameter:" option is selected, but the list below it is empty. The "Insert a new structure of the one of the following types:" option is also selected. Below this, a list of structure types is shown: "XML Schema Structure" (selected), "Database Structure", "EDI Structure", and "FlexText Structure". "OK" and "Cancel" buttons are at the bottom.

6. Select the **Directory.xsd** file in the ...\\MapForceExamples folder and click the Open button.
7. Click OK again when asked to select the root item, which should be "directory" as shown below.



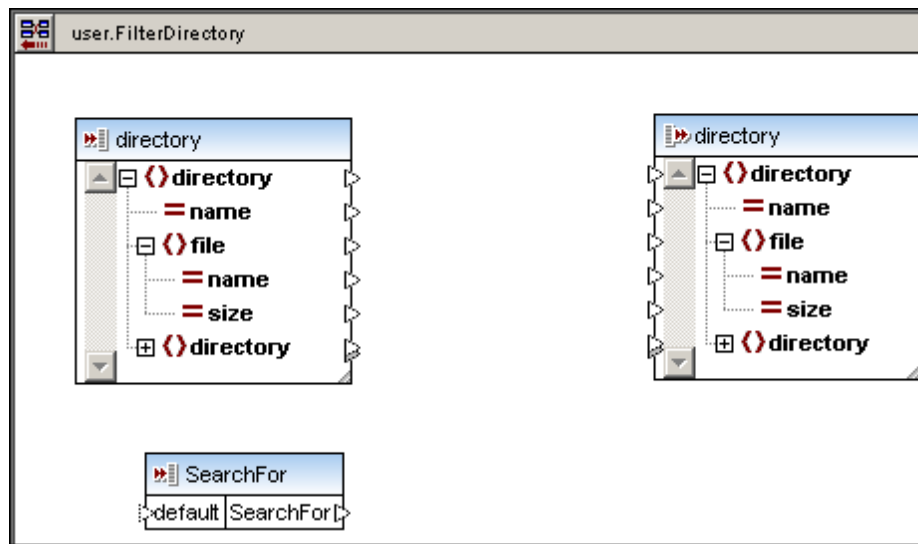


8. Click OK again to insert the complex input parameter.  
The user-defined function is shown below.



9. Delete the simple result output component, as we need to insert a complex output component here.
10. Select **Function | Insert Output...** to insert an **output** component and use the same method as outlined above, to make the output component, "directory", a complex type. You now have two complex components, one input and the other output.
11. Select **Function | Insert Input...** and insert a component of type Simple type, and enter a name e.g. **SearchFor**. Deselect the "Input is required" check box.

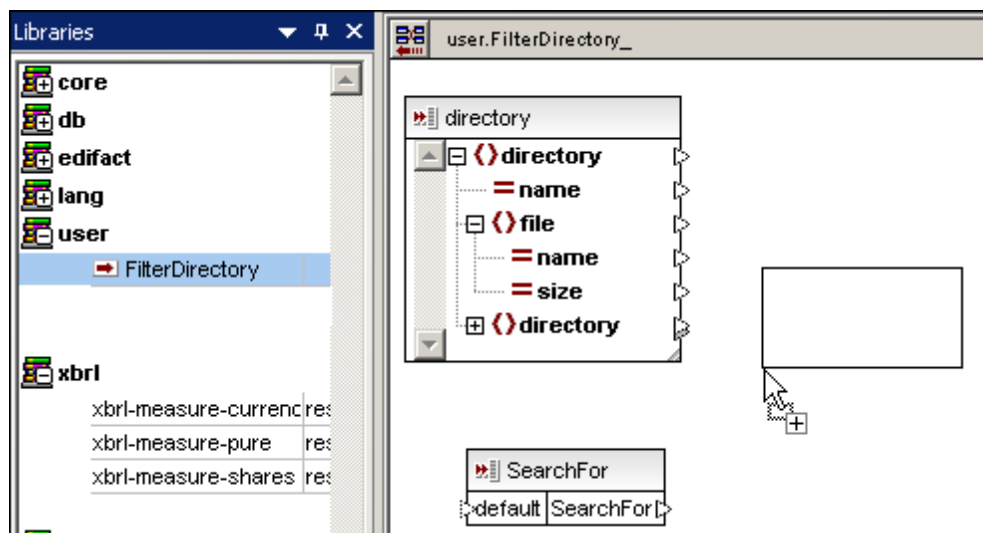




### Inserting the recursive user-defined function

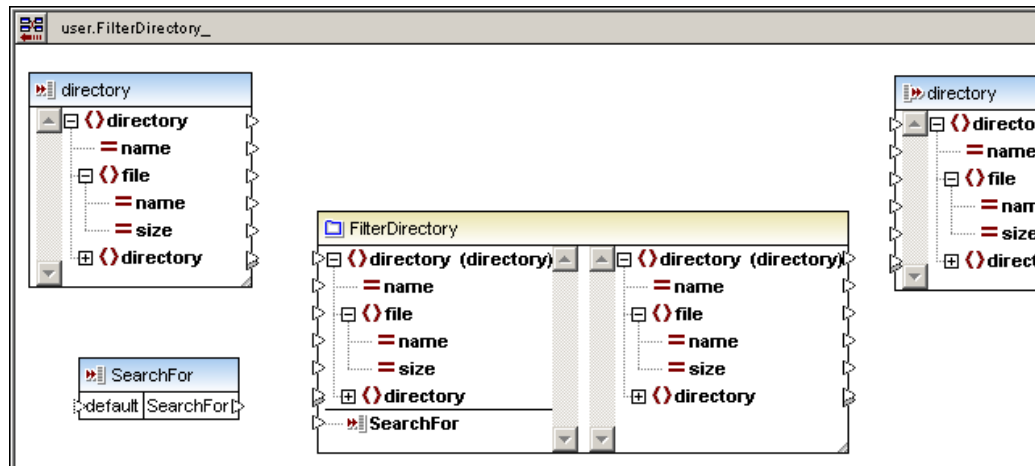
At this point, all the necessary input and output components have been defined for the user-defined function. What we need to do now is insert the "unfinished" function into the current user-defined function window. (You could do this at almost any point however.)

1. Find the FilterDirectory function in the **user** section of the **Libraries** window.
2. Click FilterDirectory then drag and drop it into the FilterDirectory window you have just been working in.

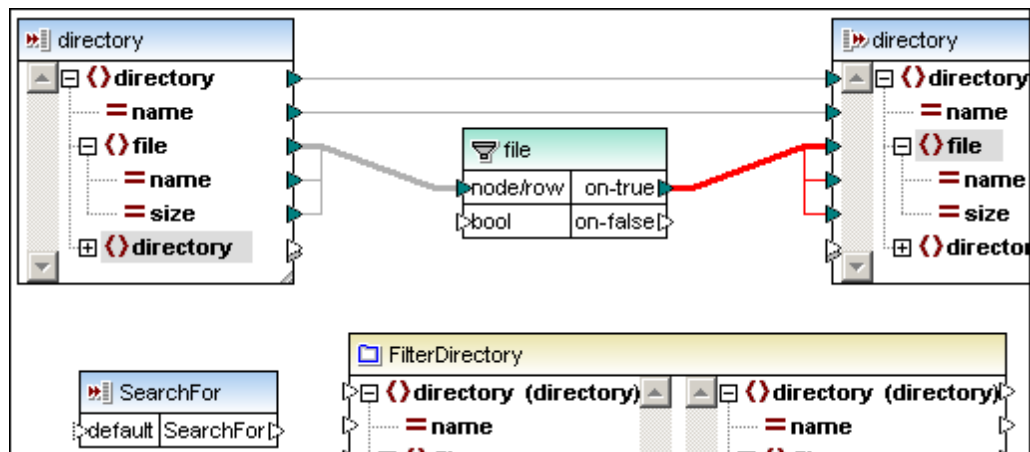


The user-defined function now appears in the user-defined function window as a recursive component.



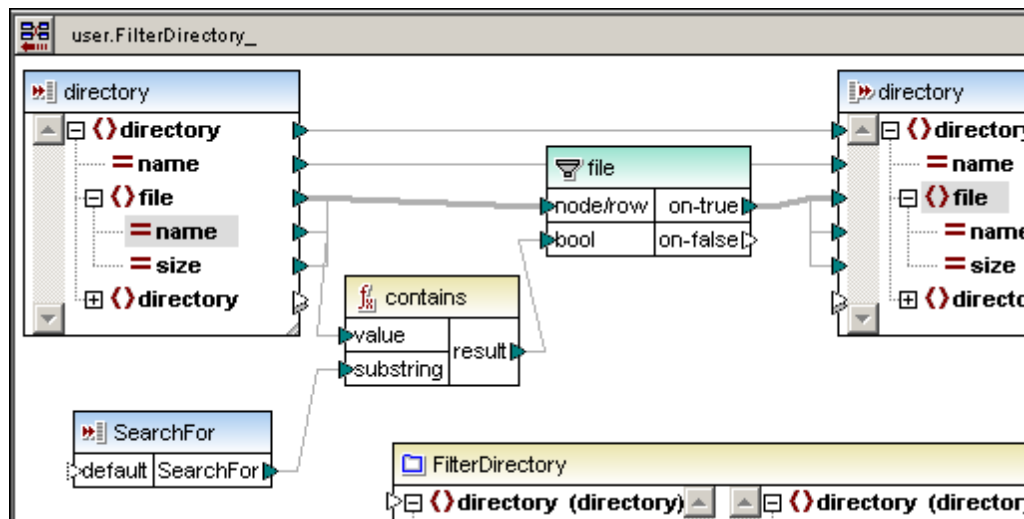


3. Connect the **directory**, **name** and **file** items of the input component to the same items in the output component.
4. Right click the connector between the **file** items and select "Insert Filter" to insert a filter component.
5. Right click the on-true connector and select **Copy-All** from the context menu. The connectors change appearance to Copy-All connectors.



6. Insert a Contains function from the **Core | String functions** library.
7. Connect **name** to **value** and the **SearchFor** parameter to **substring**, then result to the bool item of the filter.





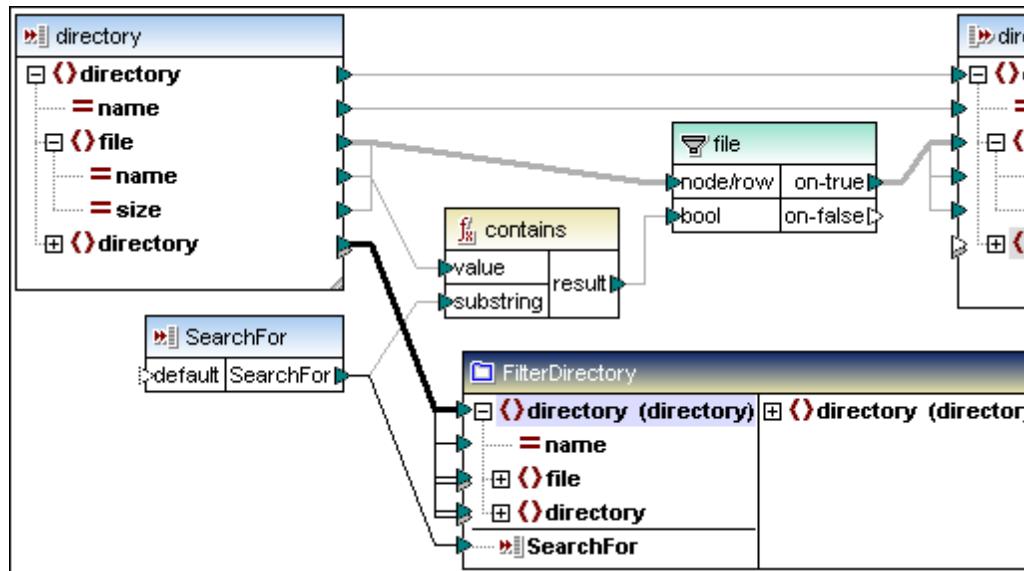
8. Connect the SearchFor item of the input component to the SearchFor item of the user-defined function.

### Defining the recursion

At this point, the mapping of a single directory recursion level is complete. Now we just need to define how to process a subdirectory.

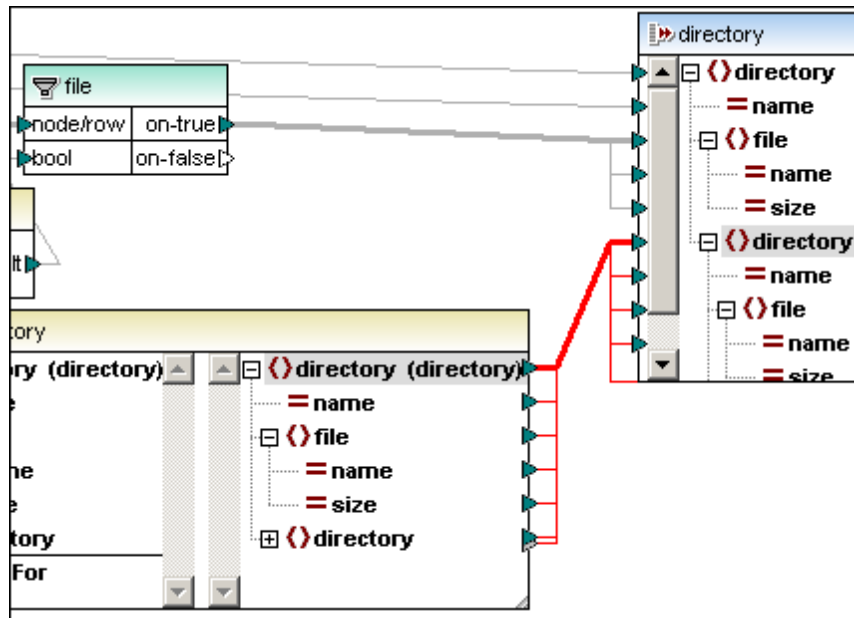
Making sure that the Toggle Autoconnect icon  is active in the icon bar:

1. Connect the lower **directory** item of the input component to the top **directory** item of the recursive user-defined function.




2. Connect the top output directory item of the user-defined function to the lower directory item of the output component.
3. Right click the top connector, select Copy-All from the context menu and click OK when prompted if you want to create Copy-All connection.



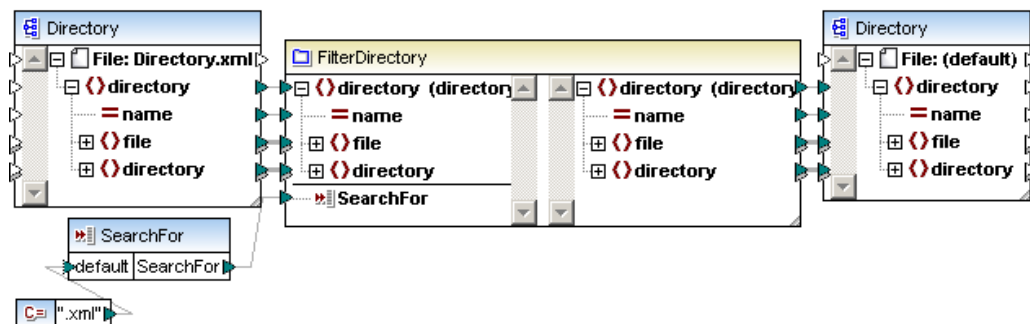


This completes the definition of the user-defined function in this window.

Click the Return to main mapping window icon,  to continue defining the mapping there.

### Main Mapping window

1. Drag the FilterDirectory function from the **user** section of the Libraries window, into the **main** mapping area.
2. Use **Insert | XML Schema file** to insert Directory.xsd and select Directory.xml as the instance file.
3. Use the same method to insert Directory.xsd and select Skip, to create the output component.
4. Insert a constant component, then a Input component e.g. SearchFor.
5. Create the connections as shown in the screenshot below.
6. When connecting the top-level connectors, directory to directory, on both sides of the user-defined component, right click the connector and select **Copy-All** from the context menu.



7. Click the Output tab to see the result of the mapping.



**Notes:**

Double clicking the lowest "directory" item in the Directory component, opens a new level of recursion, i.e. you will see a new **directory | file | directory** sublevel. Using the Copy-all connector automatically uses all existing levels of recursion in the XML instance, you do not need expand the recursion levels manually.



## 7.3 Importing Custom XSLT 1.0 or 2.0 Functions

You can extend the XSLT 1.0 and 2.0 function libraries available in MapForce with your own custom functions, provided that your custom functions return simple types.

Only custom functions that return simple data types (for example, strings) are supported.

### To import functions from an XSLT file:

1. On the **Tools** menu, click **Options**. (Alternatively, click **Add/Remove Libraries** in the lower area of the Libraries window.)
2. Next to **Libraries**, click **Add** and browse for the .xsl or .xslt file.

Imported XSLT files appear as libraries in the Libraries window, and display all named templates as functions below the library name. If you do not see the imported library, ensure you selected XSLT as transformation language (see [Selecting a Transformation Language](#)).

Note the following:

- To be eligible for import into MapForce, functions must be declared as named templates conforming to the XSLT specification in the XSLT file. You can also import functions that occur in an XSLT 2.0 document in the form `<xsl:function name="MyFunction">`. If the imported XSLT file imports or includes other XSLT files, then these XSLT files and functions will be imported as well.
- The mappable input connectors of imported custom functions depends on the number of parameters used in the template call; optional parameters are also supported.
- Namespaces are supported.
- If you make updates to XSLT files that you have already imported into MapForce, changes are detected automatically and MapForce prompts you to reload the files.
- When writing named templates, make sure that the XPath statements used in the template are bound to the correct namespace(s). To see the namespace bindings of the mapping, [preview the generated XSLT code](#).

### Datatypes in XPath 2.0

If your XML document references an XML Schema and is valid according to it, you must explicitly construct or cast datatypes that are not implicitly converted to the required datatype by an operation.

In the XPath 2.0 Data Model used by the Altova XSLT 2.0 Engine, all **atomized** node values from the XML document are assigned the `xs:untypedAtomic` datatype. The `xs:untypedAtomic` type works well with implicit type conversions.

For example,

- the expression `xs:untypedAtomic("1") + 1` results in a value of 2 because the `xdt:untypedAtomic` value is **implicitly** promoted to `xs:double` by the addition operator.
- Arithmetic operators implicitly promote operands to `xs:double`.



- Value comparison operators promote operands to `xs:string` before comparing.

#### See also:

[Example: Adding Custom XSLT 1.0 Functions](#)

[Example: Summing Node Values](#)

[XSLT 1.0 engine implementation](#)

[XSLT 2.0 engine implementation](#)

### 7.3.1 Example: Adding Custom XSLT Functions

This example illustrates how to import custom XSLT 1.0 functions into MapForce. The files needed for this example are available in the [<Documents>\Altova\MapForce2018\MapForceExamples](#) directory.

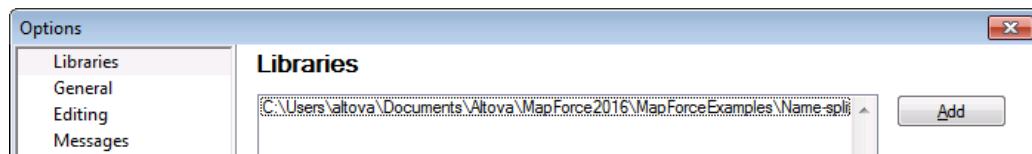
- **Name-splitter.xslt.** This XSLT file defines a named template called "**tokenize**" with a single parameter "string". The template works through an input string and separates capitalized characters with a space for each occurrence.

- **Name-splitter.xml** (the source XML instance file to be processed)
- **Customers.xsd** (the source XML schema)
- **CompletePO.xsd** (the target XML schema)

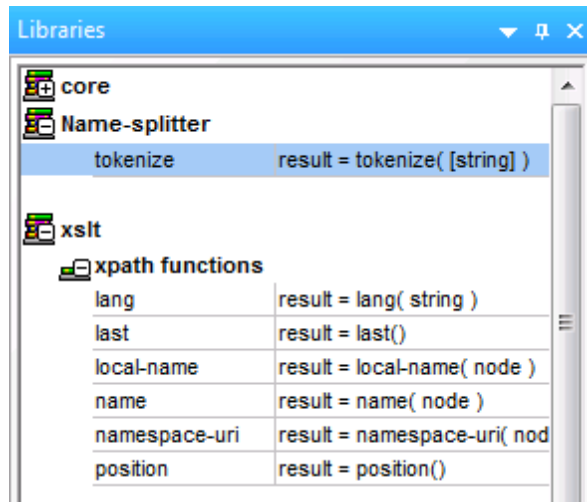
#### To add a custom XSLT function:

1. Select XSLT as transformation language (see [Selecting a Transformation Language](#)).
2. Click the **Add/Remove Libraries** button, in the lower area of the Libraries window. Alternatively, on the **Tools** menu, click **Options**, and then select **Libraries**.
3. Click **Add**, and browse for the XSL, or XSLT file, that contains the named template you want to act as a function, in this case **Name-splitter.xslt**.



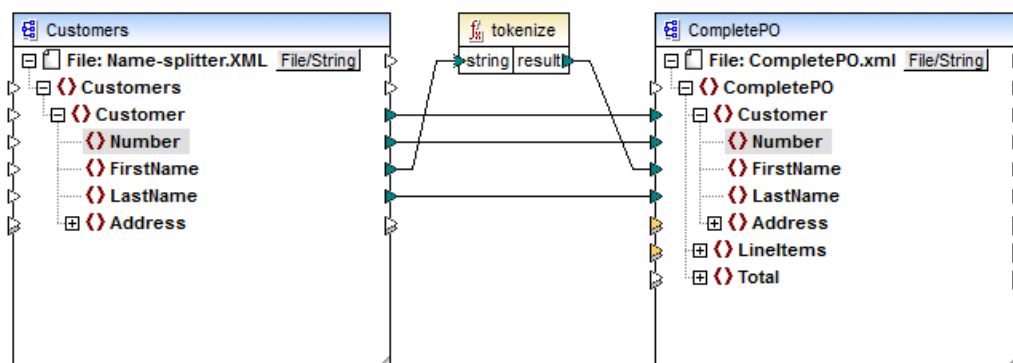


- Click **OK**. The XSLT file name appears in the Libraries window, along with the functions defined as named templates (in this example, **Name-splitter** with the **tokenize** function).



To use the XSLT function in your mapping:

- Drag the **tokenize** function into the Mapping window and map the items as show below.



- Click the XSLT tab to see the generated XSLT code.





```

<!-->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <xsl:output method="xml" encoding="UTF-8"/>
  <xsl:include href="C:\Program Files\Altova\MAPFORCE2004\MapForceExamples\Name-splitter.xslt"/>
  <xsl:template match="/Customers">
    <CompletePO>
      <xsl:attribute name="xsi:noNamespaceSchemaLocation">C:/PROGRA~1/Altova/MAPFORCE2004/MapForceExamples/Name-splitter.xslt</xsl:attribute>
      <xsl:for-each select="Customer">
        <Customer>
          <xsl:for-each select="Number">
            <Number>
              <xsl:value-of select="."/>
            </Number>
          </xsl:for-each>
          <xsl:for-each select="FirstName">
            <xsl:variable name="V47993824_47988944" select="."/>
            <xsl:variable name="V47993824_47939520">
              <xsl:call-template name="tokenize">
                <xsl:with-param name="string" select="$V47993824_47988944"/>
              </xsl:call-template>
            </xsl:variable>
          </xsl:for-each>
        </Customer>
      </xsl:for-each>
    </CompletePO>
  </xsl:template>
</xsl:stylesheet>

```

**Note:** As soon as a named template is used in a mapping, the XSLT file containing the named template is **included** in the generated XSLT code (**xsl:include href=...**), and is **called** using the command **xsl:call-template**.

3. Click the Output tab to see the result of the mapping.



```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <CompletePO xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
3    <Customer>
4      <Number>1</Number>
5      <FirstName>Fred John</FirstName>
6      <LastName>Landis</LastName>
7    </Customer>
8    <Customer>
9      <Number>2</Number>
10     <FirstName>Michelle Ann-marie</FirstName>
11     <LastName>Butler</LastName>
12   </Customer>
13   <Customer>
14     <Number>3</Number>
15     <FirstName>Ted Mac</FirstName>
16     <LastName>Little</LastName>

```

#### To remove custom XSLT libraries from MapForce:

1. Click the **Add/Remove Libraries** button, in the lower area of the Libraries window.
2. Click the XSLT library to be deleted, and then click **Delete**.



### 7.3.2 Example: Summing Node Values

This example shows you how to process multiple nodes of an XML document and have the result mapped as a single value to a target XML document. Specifically, the goal of the mapping is to calculate the price of all products in a source XML file and write it as a single value to an output XML file. The files used in this example are available in the **<Documents>\Altova\MapForce2018\MapForceExamplesTutorial\** folder:

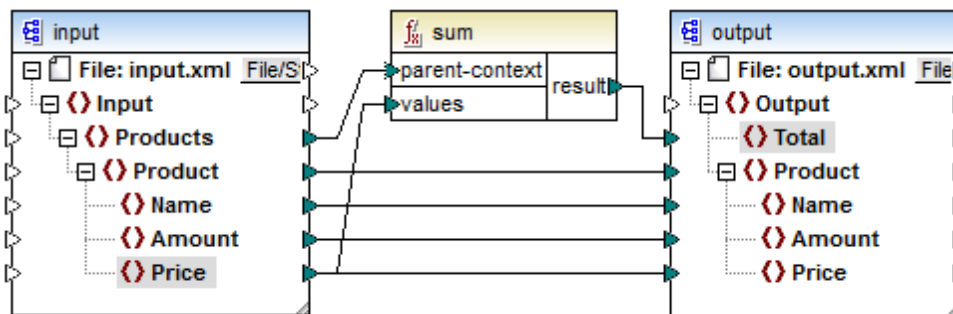
- **Summing-nodes.mfd** — the mapping file
- **input.xml** — the source XML file
- **input.xsd** — the source XML schema
- **output.xsd** — the target XML schema
- **Summing-nodes.xslt** — A custom XSLT stylesheet containing a named template to sum the individual nodes.

There are two different ways to achieve the goal of the mapping:

- By using the [sum](#) aggregate function of the **core** library. This function is available in the **Libraries** window (see also [Working with Functions](#)).
- By importing a custom XSLT stylesheet into MapForce.

#### Solution 1: Using the "sum" aggregate function

To use the **sum** aggregate function in the mapping, drag it from the **Libraries** window into the mapping. Note that the functions available in the **Libraries** window depend on the XSLT language version you selected (XSLT 1 or XSLT 2). Next, create the mapping connections as shown below.



For more information about aggregate functions of the **core** library, see also [core | aggregate functions](#).

#### Solution 2: Using a custom XSLT Stylesheet

As mentioned above, the aim of the example is to sum the `Price` fields of products in the source XML file, in this case products A and B.

```
<?xml version="1.0" encoding="UTF-8"?>
```



```
<Input xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="input.xsd">
  <Products>
    <Product>
      <Name>ProductA</Name>
      <Amount>10</Amount>
      <Price>5</Price>
    </Product>
    <Product>
      <Name>ProductB</Name>
      <Amount>5</Amount>
      <Price>20</Price>
    </Product>
  </Products>
</Input>
```

The image below shows a custom XSLT stylesheet which uses the named template "Total" and a single parameter `string`. The template works through the XML input file and sums all the values obtained by the XPath expression `/Product/Price`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

  <xsl:template match="*">
    <xsl:for-each select=".">
      <xsl:call-template name="Total">
        <xsl:with-param name="string" select="."/>
      </xsl:call-template>
    </xsl:for-each>
  </xsl:template>

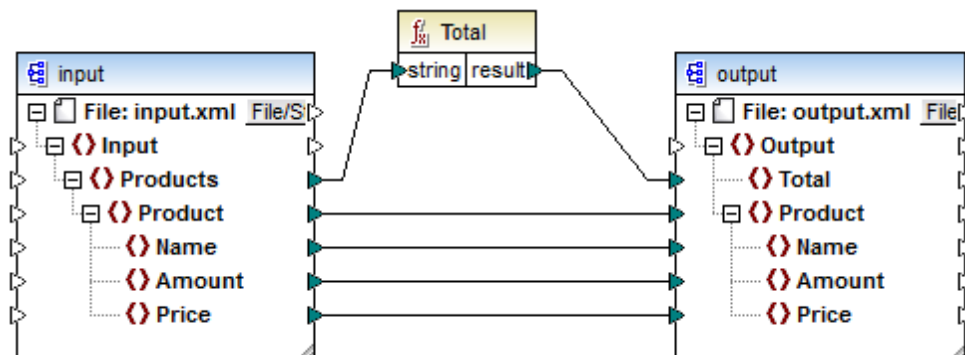
  <xsl:template name="Total">
    <xsl:param name="string"/>
    <xsl:value-of select="sum($string/Product/Price)"/>
  </xsl:template>
</xsl:stylesheet>
```

**Note:** To sum the nodes in XSLT 2.0, change the stylesheet declaration to `version="2.0"`.

To import the XSLT stylesheet into MapForce:

1. Select XSLT as transformation language. For more information, see [Selecting a Transformation Language](#).
2. In the **Libraries** window, click **Add/Remove Libraries**.
3. On the **Options** dialog box, click the **Libraries** tab.
4. Click **Add** and browse for **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\Summing-nodes.xslt**.
5. Drag the Total function from the newly created "Summing-nodes" library into the mapping, and create the mapping connections as shown below.





To preview the mapping result, click the **Output** tab. The sum of the two `Price` fields is now displayed in the `Total` field.

```
<?xml version="1.0" encoding="UTF-8"?>
<Output xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="output.xsd">
  <Total>25</Total>
  <Product>
    <Name>ProductA</Name>
    <Amount>10</Amount>
    <Price>5</Price>
  </Product>
  <Product>
    <Name>ProductB</Name>
    <Amount>5</Amount>
    <Price>20</Price>
  </Product>
</Output>
```



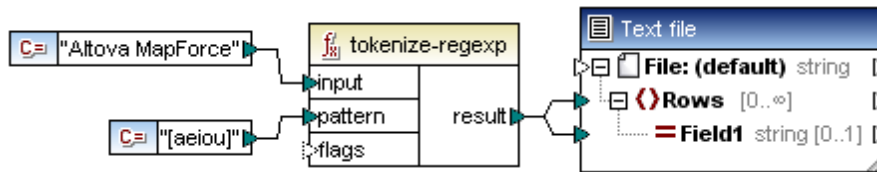
## 7.4 Regular Expressions

MapForce can use regular expressions in the **pattern** parameter of the [tokenize-regex](#) function, to find specific strings.

The regular expression syntax and semantics for XSLT and XQuery are identical to those defined in <https://www.w3.org/TR/xmlschema-2/>. Please note that there are slight differences in regular expression syntax between the various programming languages.

### Terminology

input	the string that the regex works on
pattern	the regular expression
flags	optional parameter to define how the regular expression is to be interpreted
result	the result of the function



Tokenize-regex returns a sequence of strings. The connection to the Rows item creates one row per item in the sequence.

### regex syntax

**Literals** e.g. a single character:

e.g. The letter "a" is the most basic regex. It matches the first occurrence of the character "a" in the string.

### Character classes []

This is a set of characters enclosed in square brackets.

**One**, and only one, of the characters in the square brackets are matched.

pattern **[aeiou]**

Matches a lowercase vowel.

pattern **[mj]ust**

Matches must or just

Please note that "pattern" is case sensitive, a lower case **a** does not match the uppercase **A**.

### Character ranges [a-z]

Creates a range between the two characters. Only one of the characters will be matched at one time.



pattern **[a-z]**

Matches any lowercase characters between a and z.

*negated classes* **[^]**

using the caret as the first character after the opening bracket, negates the character class.

pattern **[^a-z]**

Matches any character not in the character class, including newlines.

### Meta characters **"."**

Dot meta character

matches **any single** character (except for newline)

pattern **.**

Matches any single character.

### Quantifiers **? + \* {}**

Quantifiers define how often a regex component must repeat within the input string, for a match to occur.

**?**

zero or one                      preceding string/chunk is optional

**+**

one or more                      preceding string/chunks may match one or more times

**\***

zero or more                      preceding string/chunks may match zero or more times

**{}**

min / max                      no. of repetitions a string/chunks has to match  
repetitions

e.g. `mo{1,3}` matches `mo`, `moo`, `moou`.

**()**

subpatterns

parentheses are used to group parts of a regex together.

**|**

Alternation/or      allows the testing of subexpressions from left to right.

**(horse|make) sense** - will match "horse sense" or "make sense"

### Flags

These are optional parameters that define how the regular expression is to be interpreted.

Individual letters are used to set the options, i.e. the character is present. Letters may be in any order and can be repeated.

**s**



If present, the matching process will operate in the "dot-all" mode.

The meta character "." matches any character whatsoever. If the input string contains "hello" and "world" on two *different* lines, the regular expression "hello\*world" will only match if the **s** flag/character is set.

### ***m***

If present, the matching process operates in multi-line mode.

In multi-line mode the caret ^ matches the start of **any** line, i.e. the start of the entire string **and** the first character after a newline character.

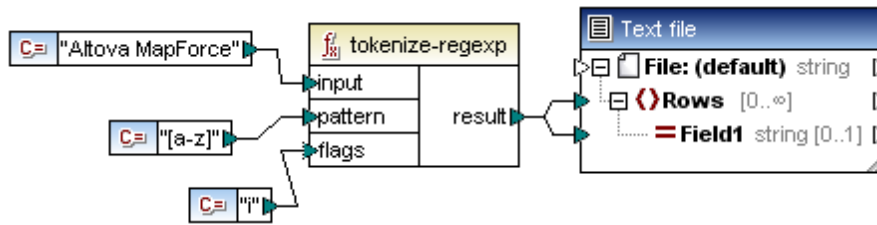
The dollar character \$ matches the end of **any** line, i.e. the end of the entire string and the character immediately before a newline character.

Newline is the character #x0A.

### ***i***

If present, the matching process operates in case-insensitive mode.

The regular expression [a-z] plus the **i** flag would then match all letters a-z and A-Z.



### ***x***

If present, whitespace characters are removed from the regular expression prior to the matching process. Whitespace chars. are #x09, #x0A, #x0D and #x20.

*Exception:*

Whitespace characters within character class expressions are not removed e.g. [#x20].

Please note:

When generating code, the advanced features of the regex syntax might differ slightly between the various languages, please see the specific regex documentation for your language.



## 7.5 Function Library Reference

This reference chapter describes the MapForce built-in functions available in the Libraries pane, organized by library.

The availability of function libraries in the Libraries pane depends on the transformation language you have selected (see [Selecting a transformation language](#) ).

**XPath 2.0 restrictions:** Several XPath 2.0 functions dealing with sequences are currently not available.

### 7.5.1 core | aggregate functions

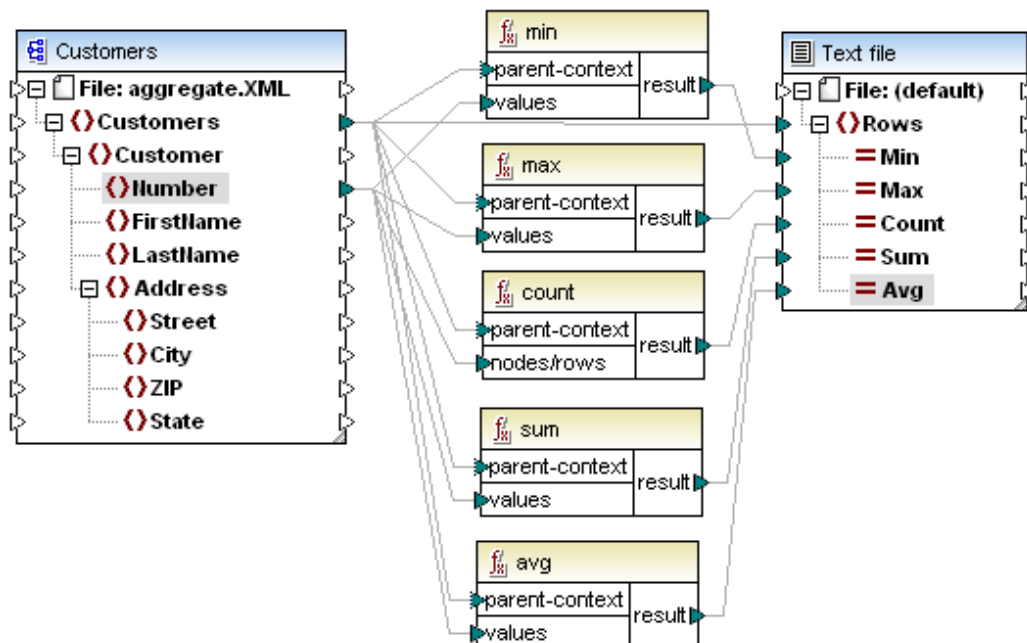
Aggregate functions perform operations on a set, or sequence, of input values. The input data for min, max, sum and avg is converted to the **decimal** data type for processing.

- The input values must be connected to the **values** parameter of the function.
- A context node (item) can be connected to the **parent-context** parameter to override the default context from which the input sequence is taken. The parent-context parameter is optional.
- The **result** of the function is connected to the specific target item.

The mapping shown below is available as **Aggregates.mfd** in the ...Tutorial folder and shows how these functions are used.

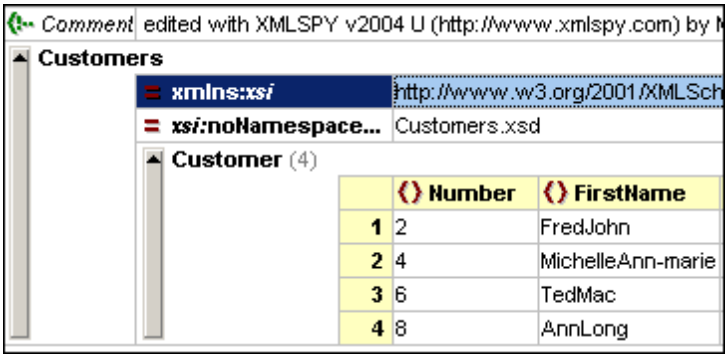
Aggregate functions have two input items.

- **values** (nodes/rows) is connected to the source item that provides the data, in this case Number.
- **parent-context** is connected to the item you want to iterate over, i.e. the context, in this case over all Customers. The parameter is, however, optional.





The input instance in this case is an XML file containing the following data:



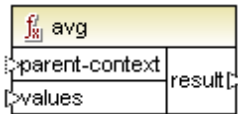
- The source data supplied to the values item is the number sequence 2,4,6,8.
  - The output component in this case is a simple text file.
- Clicking the Output tab for the above mapping delivers the following result:

1	2,8,4,20,5
2	

min=2, max=8, count=4, sum=20 and avg=5.

7.5.1.1 avg

Returns the average value of all values within the input sequence. The average of an empty set is an empty set. Not available in XSLT1.



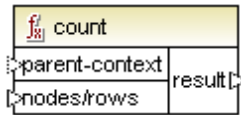
Argument	Description
parent-context	Optional argument. Supplies the <a href="#">parent context</a> . See also <a href="#">Overriding the Mapping Context</a> .
values	This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric.

For an example of usage, see the mapping **GroupTemperaturesByYear.mfd** in the **<Documents>\Altova\MapForce2018\MapForceExamples\** directory.

7.5.1.2 count

Returns the number of individual items making up the input sequence. The count of an empty set is zero. Limited functionality in XSLT1.

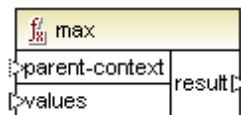




Argument	Description
<b>parent-context</b>	Optional argument. Supplies the <a href="#">parent context</a> . See also <a href="#">Overriding the Mapping Context</a> .
<b>nodes/rows</b>	This argument must be connected to the source item to be counted.

### 7.5.1.3 *max*

Returns the maximum value of all numeric values in the input sequence. Note that this function returns an empty set if the **strings** argument is an empty set. Not available in XSLT1.

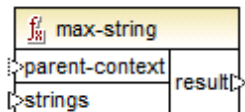


Argument	Description
<b>parent-context</b>	Optional argument. Supplies the <a href="#">parent context</a> . See also <a href="#">Overriding the Mapping Context</a> .
<b>values</b>	This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric. To get the maximum from a sequence of strings, use the <a href="#">max-string</a> function.

For an example of usage, see the mapping **GroupTemperaturesByYear.mfd** in the **<Documents>\Altova\MapForce2018\MapForceExamples\** directory.

### 7.5.1.4 *max-string*

Returns the maximum value of all string values in the input sequence. For example, `max-string("a", "b", "c")` returns "c". This function is not available in XSLT1.



Argument	Description
<b>parent-context</b>	Optional argument. Supplies the <a href="#">parent context</a> . See also <a href="#">Overriding the Mapping Context</a> .

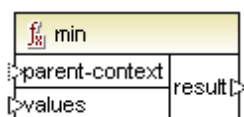


Argument	Description
<b>strings</b>	This argument must be connected to a source item which supplies the actual data. The supplied argument value must be a sequence (zero or many) of <code>xs:string</code> .

Note that the function returns an empty set if the **strings** argument is an empty set.

### 7.5.1.5 *min*

Returns the minimum value of all numeric values in the input sequence. The minimum of an empty set is an empty set. Not available in XSLT1.

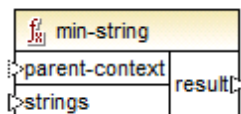


Argument	Description
<b>parent-context</b>	Optional argument. Supplies the <a href="#">parent context</a> . See also <a href="#">Overriding the Mapping Context</a> .
<b>values</b>	This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric. To get the minimum from a sequence of strings, use the <a href="#">min-string</a> function.

For an example of usage, see the mapping **GroupTemperaturesByYear.mfd** in the **<Documents>\Altova\MapForce2018\MapForceExamples\** directory.

### 7.5.1.6 *min-string*

Returns the minimum value of all string values in the input sequence. For example, `min-string("a", "b", "c")` returns "a". This function is not available in XSLT1.



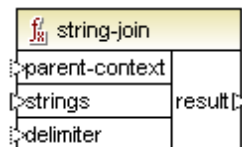
Argument	Description
<b>parent-context</b>	Optional argument. Supplies the <a href="#">parent context</a> . See also <a href="#">Overriding the Mapping Context</a> .
<b>strings</b>	This argument must be connected to a source item which supplies the actual data. The supplied argument value must be a sequence (zero or many) of <code>xs:string</code> .



Note that the function returns an empty set if the **strings** argument is an empty set.

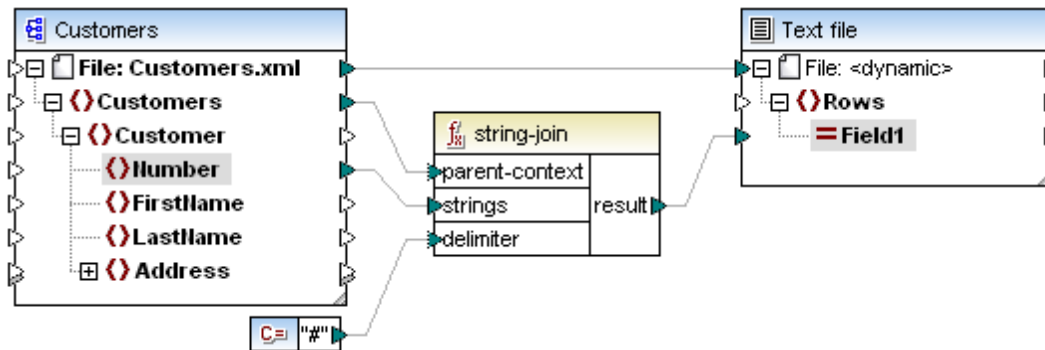
### 7.5.1.7 *string-join*

Concatenates all the values of the input sequence into one string delimited by whatever string you choose to use as the delimiter. The string-join of an empty set is the empty string. Not available in XSLT1.



The example below contains four separate customer numbers 2 4 6 and 8. The constant character supplies a hash character "#" as the delimiter.

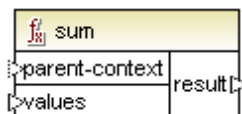
Result = 2#4#6#8



If you do not supply a delimiter, then the default is an empty string, i.e. no delimiter of any sort.  
Result = 2468.

### 7.5.1.8 *sum*

Returns the arithmetic sum of all values in the input sequence. The sum of an empty set is zero.



Argument	Description
<b>parent-context</b>	Optional argument. Supplies the <a href="#">parent context</a> . See also <a href="#">Overriding the Mapping Context</a> .
<b>values</b>	This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric.

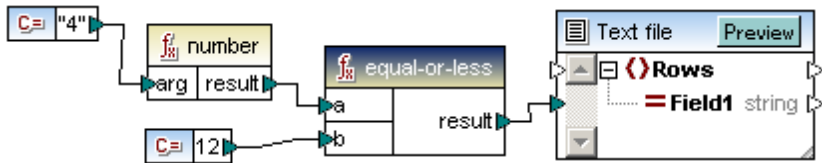
See also [Example: Summing Node Values](#).



7.5.2 core | conversion functions

To support explicit data type conversion, several type conversion functions are available in the **conversion** library. Note that, in most cases, MapForce creates necessary conversions automatically and these functions need to be used only in special cases.

If the input nodes are of differing types, e. g. integer and string, you can use the conversion functions to force a string or numeric comparison.



In the example above the first constant is of type string and contains the string "4". The second constant contains the numeric constant 12. To be able to compare the two values explicitly the types must agree.

Adding a **number** function to the first constant converts the string constant to the numeric value of 4. The result of the comparisons is then "true".

Note that if the number function were not be used, i.e 4 would be connected directly to the **a** parameter, a string compare would occur, with the result being false.

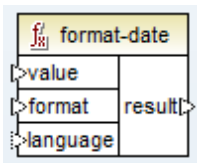
7.5.2.1 boolean

Converts an input numeric value into a boolean (as well as a string to numeric - true to 1). E.g. 0 to "false", or 1 to "true", for further use with logical functions (equal, greater etc.) filters, or if-else functions.



7.5.2.2 format-date

Converts an `xs:date` input value into a string and formats it according to specified options.

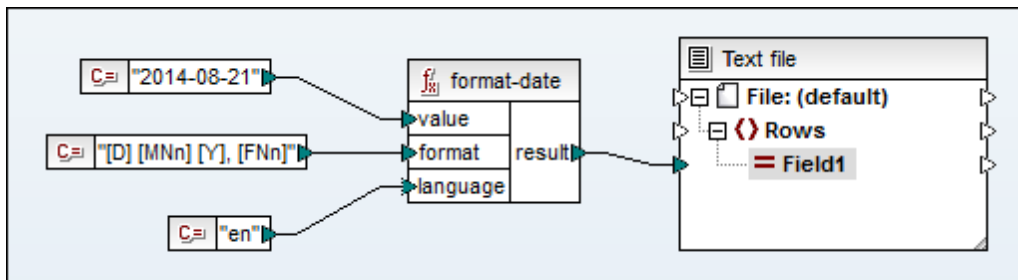


Argument	Description
value	The date to be formatted.



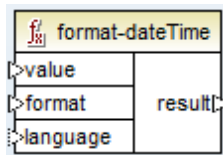
Argument	Description
format	A format string identifying the way in which the date is to be formatted. This argument is used in the same way as the <code>format</code> argument in the <code>format-dateTime</code> function.
language	Optional argument. When supplied, the name of the month and the day of the week are returned in a specific language. Valid values: <div style="margin-left: 20px;"> <b>en (default)</b>      English  <b>es</b>                      Spanish  <b>de</b>                      German  <b>ja</b>                      Japanese </div>

In the following example, the output result is: "21 August 2014, Thursday". To translate this value to Spanish, set the value of the `language` argument to `es`.



### 7.5.2.3 *format-dateTime*

Converts a date and time value (`xs:dateTime`) into a string. The string representation of date and time is formatted according to the value of the `format` argument.



Argument	Description
value	The <code>xs:dateTime</code> value to be formatted.
format	A format string identifying the way in which <b>value</b> is to be formatted.
language	Optional argument. When supplied, the name of the month and the day of the week are returned in a specific language. Valid values: <div style="margin-left: 20px;"> <b>en (default)</b>      English </div>



Argument	Description
<b>es</b>	Spanish
<b>de</b>	German
<b>ja</b>	Japanese

**Note:** If the function's output (result) is connected to a node of type other than string, the formatting may be lost as the value is cast to the target type. This automatic cast can be disabled by unchecking the **Cast target values to target types** check box in the Component Settings of the target component (see [Changing the Component Settings](#)).

The **format** argument consists of a string containing so-called variable markers enclosed in square brackets. Characters outside the square brackets are literal characters to be copied into the result. If square brackets are needed as literal characters in the result, then they should be doubled.

Each variable marker consists of a component specifier identifying which component of the date or time is to be displayed, an optional formatting modifier, another optional presentation modifier and an optional width modifier, preceded by a comma if it is present.

```
format := (literal | argument)*
argument := [component(format)?(presentation)?(width)?]
width := , min-width ("-" max-width)?
```

The components are as follows:

Specifier	Description	Default Presentation
Y	year (absolute value)	four digits (2010)
M	month of the year	1-12
D	day of month	1-31
d	day of year	1-366
F	day of week	name of the day (language dependent)
W	week of the year	1-53
w	week of month	1-5
H	hour (24 hours)	0-23
h	hour (12 hour)	1-12
P	A.M. or P.M.	alphabetic (language dependent)
m	minutes in hour	00-59
s	seconds in minute	00-59
f	fractional seconds	numeric, one decimal place



Specifier	Description	Default Presentation
Z	timezone as a time offset from UTC	+08:00
z	timezone as a time offset using GMT	GMT+n

The formatting modifier can be one of the following:

Character	Description	Example
1	decimal numeric format with no leading zeros: 1, 2, 3, ...	1, 2, 3
01	decimal format, two digits: 01, 02, 03, ...	01, 02, 03
N	name of component, upper case	MONDAY, TUESDAY <sup>1)</sup>
n	name of component, lower case	monday, tuesday <sup>1)</sup>
Nn	name of component, title case	Monday, Tuesday <sup>1)</sup>

**Note:** N, n, and Nn modifiers only support the following components: M, d, D.

The width modifier, if present, is introduced by a comma. It takes the form:

, min-width ("-" max-width)?

The table below illustrates some examples of formatting `xs:dateTime` values with the help of the `format-dateTime` function. The "Value" column specifies the value supplied to the `value` argument. The "Format" column specifies the value of the `format` argument. The "Result" column illustrates what is returned by the function.

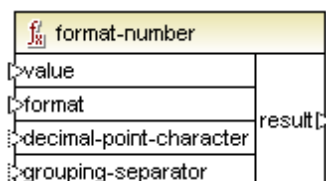
Value	Format	Result
2003-11-03T00:00:00	[D]/[M]/[Y]	3/11/2003
2003-11-03T00:00:00	[Y]-[M,2]-[D,2]	2003-11-03
2003-11-03T00:00:00	[Y]-[M,2]-[D,2] [H,2]:[m]:[s]	2003-11-03 00:00:00
2010-06-02T08:02	[Y] [MNn] [D01] [F,3-3] [d] [H]:[m]:[s].[f]	2010 June 02 Wed 153 8:02:12.054
2010-06-02T08:02	[Y] [MNn] [D01] [F,3-3] [d] [H]:[m]:[s].[f] [z]	2010 June 02 Wed 153 8:02:12.054 GMT+02:00
2010-06-02T08:02	[Y] [MNn] [D1] [F] [H]:[m]:[s].[f] [Z]	2010 June 2 Wednesday 8:02:12.054 +02:00
2010-06-	[Y] [MNn] [D] [F,3-3] [H01]:[m]:[s]	2010 June 2 Wed



Value	Format	Result
02T08:02		08:02:12

### 7.5.2.4 *format-number*

Converts a number into a string. The function is available for XSLT 1.0, XSLT 2.0, Java, C#, C++ and Built-in execution engine.



Argument	Description
value	Mandatory argument. Supplies the number to be formatted.
format	Mandatory argument. Supplies a format string that identifies the way in which the number is to be formatted. This argument is used in the same way as the <code>format</code> argument in the <b>format-dateTime</b> function.
decimal-point-format	Optional argument. Supplies the character to be used as the decimal point character. The default value is the full stop ( . ) character.
grouping-separator	Optional argument. Supplies the character used to separate groups of numbers. The default value is the comma ( , ) character.

**Note:** If the function's output (i.e. result) is connected to a node of type other than string, the formatting may be lost as the value is cast to the target type. This automatic cast can be disabled by unchecking the **Cast target values to target types** check box in the component settings of the target component.

Format:

```

format := subformat (;subformat)?
subformat := (prefix)? integer (.fraction)? (suffix)?
prefix := any characters except special characters
suffix := any characters except special characters
integer := (#)* (0)* ( allowing ',' to appear)
fraction := (0)* (#)* (allowing ',' to appear)

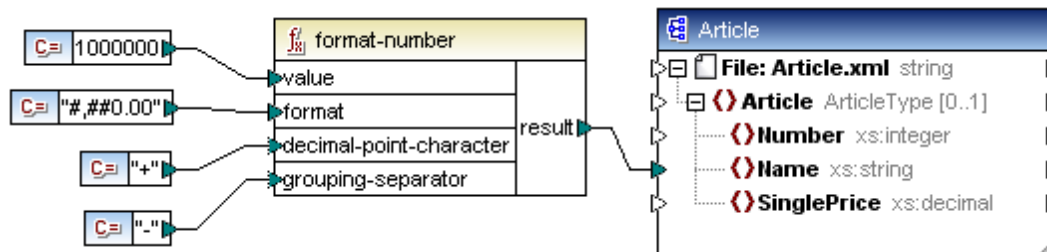
```

The first *subformat* is used for formatting positive numbers, and the second subformat for negative numbers. If only one *subformat* is specified, then the same subformat will be used for negative numbers, but with a minus sign added before the *prefix*.



Special Character	default	Description
zero-digit	0	A digit will always appear at this point in the result
digit	#	A digit will appear at this point in the result string unless it is a redundant leading or trailing zero
decimal-point	.	Separates the integer and the fraction part of the number.
grouping-separator	,	Separates groups of digits.
percent-sign	%	Multiplies the number by 100 and shows it as a percentage.
per-mille	‰	Multiplies the number by 1000 and shows it as per-mille.

The characters used for decimal-point-character and grouping-separator are always "." and ",", respectively. They can, however, be changed in the formatted output, by mapping constants to these nodes.



The result of the format number function shown above.

- The decimal-point character was changed to a "+".
- The grouping separator was changed to a "-".

```
<Article xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'>
  <Number xs:integer>1-000-000+00</Number>
</Article>
```

## Rounding

The rounding method used for this function is "half up", e.g. the value gets rounded up if the fraction is greater than or equal to 0.5. The value gets rounded down if the fraction is less than 0.5. This method of rounding only applies to generated code and the built-in execution engine.

In XSLT 1.0, the rounding mode is undefined. In XSLT 2.0, the rounding mode is "round-half-to-even".

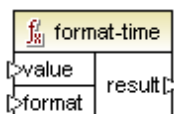
Number	Format String	Result
1234.5	#,##0.00	1,234.50



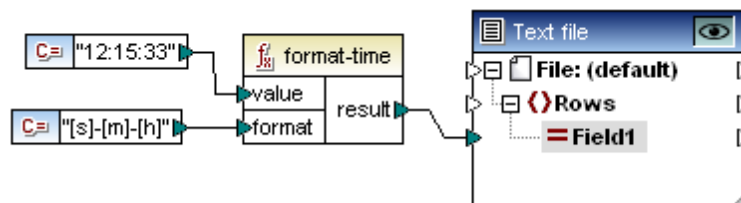
Number	Format String	Result
123.456	#,##0.00	123.46
1000000	#,##0.00	1,000,000.00
-59	#,##0.00	-59.00
1234	###0.0###	1234.0
1234.5	###0.0###	1234.5
.00025	###0.0###	0.0003
.00035	###0.0###	0.0004
0.25	#00%	25%
0.736	#00%	74%
1	#00%	100%
-42	#00%	-4200%
-3.12	#.00;(#.00)	(3.12)
-3.12	#.00;#.00CR	3.12CR

### 7.5.2.5 *format-time*

Converts an xs:time input value into a string. The `format` argument is used in the same way as the `format` argument in the **format-dateTime** function.



E.g

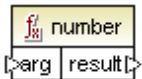


Result: 33-15-12

### 7.5.2.6 *number*

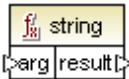
Converts an input string into a number. Also converts a boolean input to a number.





### 7.5.2.7 *string*

Converts an input value into a string. The function can also be used to retrieve the text content of a node.



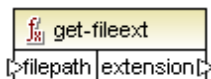
If the input node is a XML complex type, then all descendents are also output as a single string.

## 7.5.3 **core | file path functions**

The **file path** functions allow you to directly access and manipulate file path data, i.e. folders, file names, and extensions for further processing in your mappings. They can be used in all languages supported by MapForce.

### 7.5.3.1 *get-fileext*

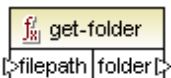
Returns the extension of the file path including the dot "." character.



E.g. 'c:\data\Sample.mfd' returns '.mfd'

### 7.5.3.2 *get-folder*

Returns the folder name of the file path including the trailing slash, or backslash character.



E.g. 'c:/data/Sample.mfd' returns 'c:/data/'

### 7.5.3.3 *main-mfd-filepath*

Returns the full path of the mfd file containing the main mapping. An empty string is returned if the mfd is currently unsaved.





### 7.5.3.4 *mfd-filepath*

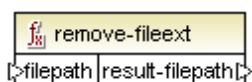
If the function is called in the main mapping, it returns the same as main-mfd-filepath function, i.e. the full path of the mfd file containing the main mapping. An empty string is returned if the mfd is currently unsaved.



If called within an **user-defined function** which is **imported** by a mfd-file, it returns the full path of the imported mfd file which contains the **definition** of the user-defined function.

### 7.5.3.5 *remove-fileext*

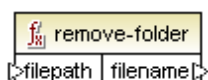
Removes the extension of the file path including the dot-character.



E.g. 'c:/data/Sample.mfd' returns 'c:/data/Sample'.

### 7.5.3.6 *remove-folder*

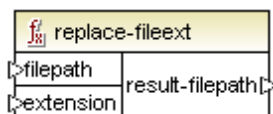
Removes the directory of the file path including the trailing slash, or backslash character.



E.g. 'c:/data/Sample.mfd' returns 'Sample.mfd'.

### 7.5.3.7 *replace-fileext*

Replaces the extension of the file path supplied by the filepath parameter, with the one supplied by the connection to the extension parameter.

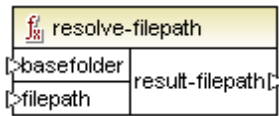


E.g. c:/data/Sample.**mfd**' as the input filepath, and '**mfp**' as the extension, returns 'c:/data/Sample.mfp'

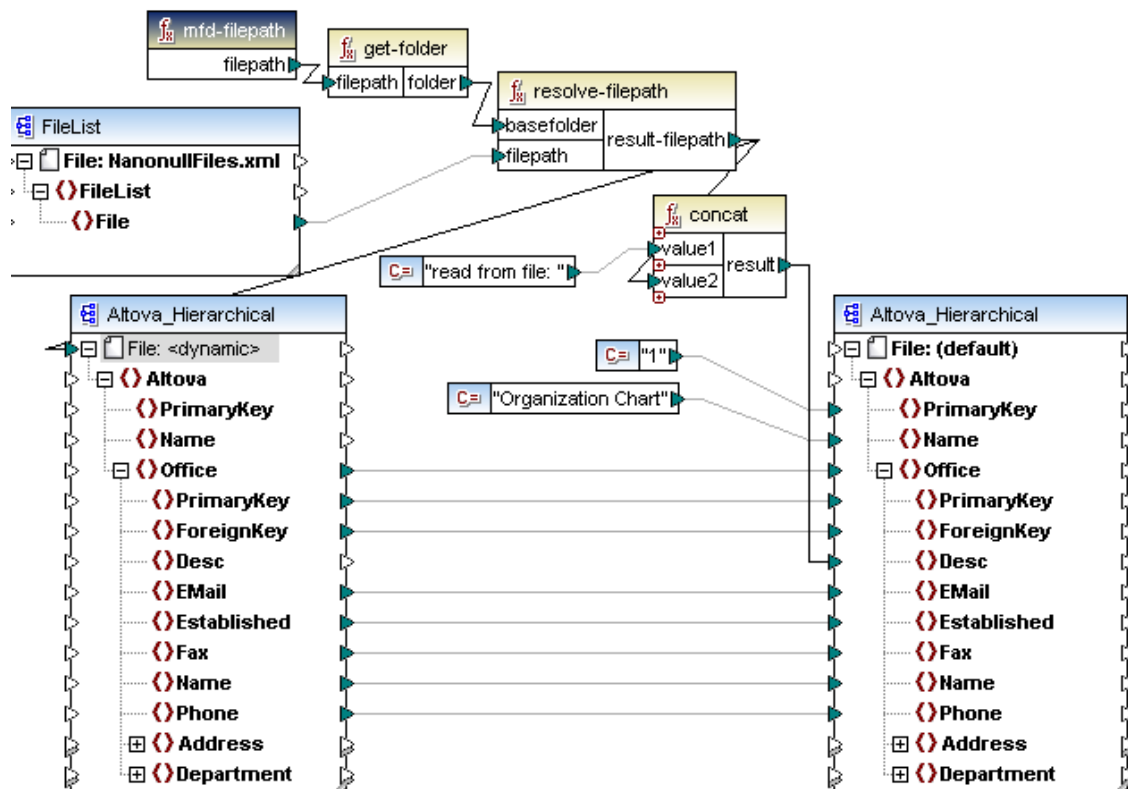


### 7.5.3.8 *resolve-filepath*

Resolves a relative file path to a relative, or absolute, base folder. The function supports '.' (current directory) and '..' (parent directory).



For an example, see the mapping **MergeMultipleFiles\_List.mfd** available in the ... \MapForceExamples folder.



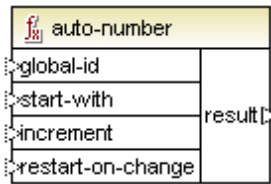
## 7.5.4 core | generator functions

The **core / generator** functions library includes functions which generate values.

### 7.5.4.1 *auto-number*

The **auto-number** function generates integers in target nodes of a component, depending on the various parameters you define. The function result is a value starting at **start\_with** and increased by **increment**. Default values are: start-with=1 and increase=1. Both parameters can be negative.





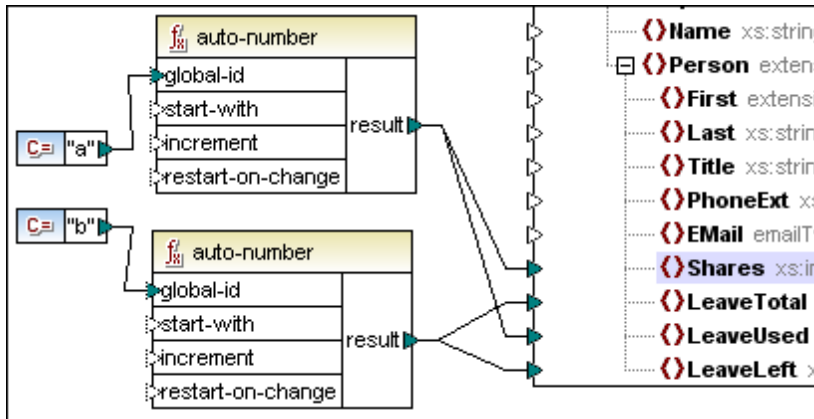
Make sure that the result connector (of the auto-number function) is **directly** connected to a target node. The exact order in which functions are called by the generated mapping code is undefined. MapForce may choose to cache calculated results for reuse, or evaluate expressions in any order. It is therefore strongly recommended to take care when using the auto-number function.

### global-id

This parameter allows you to synchronize the number sequence output of two separate auto-number functions connected to a single target component.

If the two auto-number functions do **not** have the same global-id, then each increments the target items separately. In the example below, each function has a different global-id i.e. a and b.

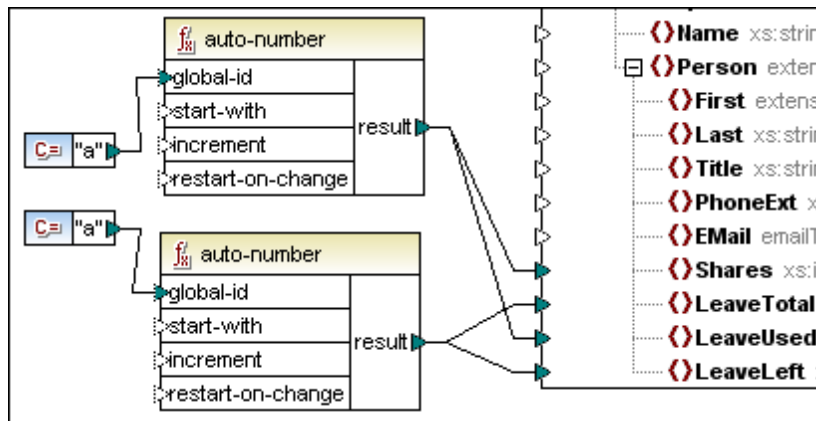
The output of the mapping is 1,1,2,2. The top function supplies the first 1 and the lower one the second 1.



If both functions have **identical** global-ids, **a** in this case, then each function "knows" about the current auto-number state (or actual value) of the other, and both numbers are then synchronised/ in sequence.

The output of the mapping is therefore 1, 2, 3, 4. The top function supplies the first 1 and the lower one now supplies a 2.



**start-with**

The initial value used to start the auto numbering sequence. Default is 1.

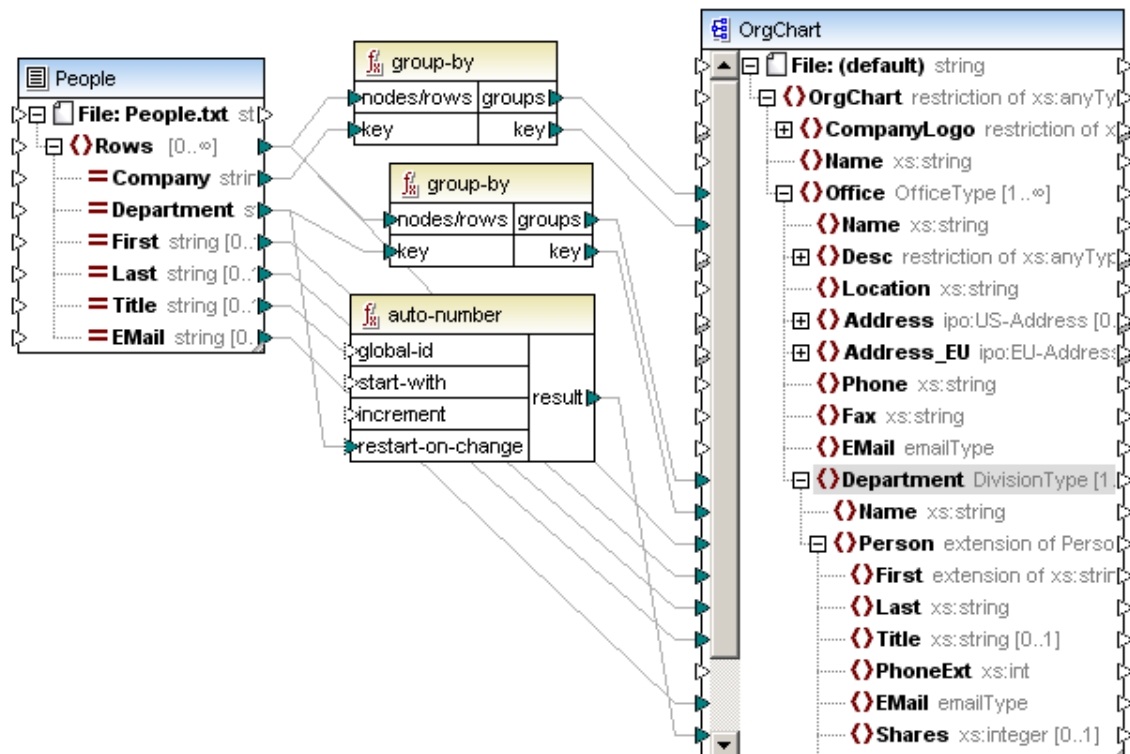
**increment**

The increment you want auto-number sequence to increase by. Default is 1.

**restart on change**

Resets the auto-number counter to "start-with", when the **content** of the connected item changes.

In the example below, start-with and increment are both using the default 1. As soon as the **content** of Department changes, i.e. the department name changes, the counter is reset and starts at 1 for each new department.





## 7.5.5 core | logical functions

Logical functions are (generally) used to compare input data with the result being a boolean "true" or "false". They are generally used to test data before passing on a subset to the target component using a filter.

input parameters = **a | b**, or **value1 | value2**  
 output parameter = result

The evaluation result of two input nodes depends on the input values as well as the data types used for the comparison.

For example, the 'less than' comparison of the integer values 4 and 12 yields the boolean value "true", since 4 is less than 12. If the two input strings contain '4' and '12', the lexical analysis results in the output value "false", since '4' is alphabetically greater than the first character '1' of the second operand (12).

If all input data types are of the same type, e.g. all input nodes are numerical types, or strings, then the comparison is done for the common type.

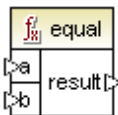
If the input nodes are of differing types (for example, integer and string, or string and date), then the data type used for the comparison **is the most general (least restrictive) input data type** of the two input types.

Before comparing two values, all input values are converted to a common datatype. Using the previous example; the datatype "string" is less restrictive than "integer". Comparing integer value 4 with the string '12', converts integer value 4 to the string '4', which is then compared with the string '12'.

**Note:** Logical functions cannot be used to test the existence of null values. If you supply a null value as argument to a logical function, it returns a null value. For more information about handling null values, see [Nil Values / Nillable](#).

### 7.5.5.1 *equal*

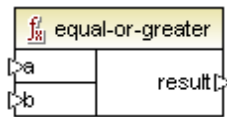
Result is true if a=b, else false.



### 7.5.5.2 *equal-or-greater*

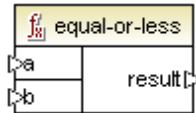
Result is true if a is equal/greater than b, else false.





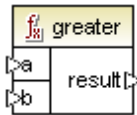
### 7.5.5.3 *equal-or-less*

Result is true if a is equal/less than b, else false.



### 7.5.5.4 *greater*

Result is true if a is greater than b, else false.



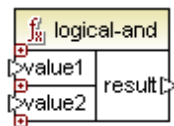
### 7.5.5.5 *less*

Result is true if a is less than b, else false.

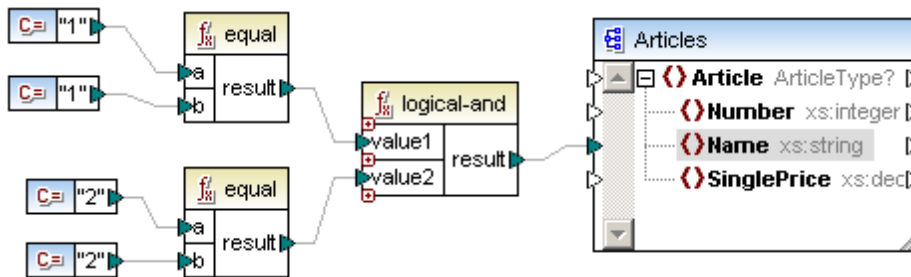


### 7.5.5.6 *logical-and*

If **both** value1 and value2 of the logical-and function are **true**, then result is true; if different then false.

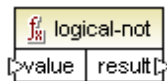




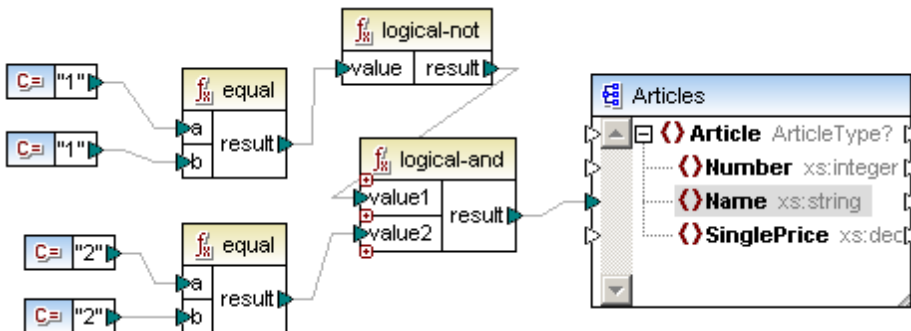


### 7.5.5.7 *logical-not*

Inverts or flips the logical state/result; if input is true, result of logical-not function is false. If input is false then result is true.

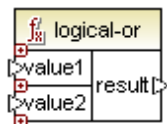


The logical-not function shown below, inverts the result of the equal function. The logical-and function now only returns true if boolean values of value1 and value2 are different, i.e. true-false, or false-true.



### 7.5.5.8 *logical-or*

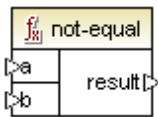
Requires both input values to be boolean. If **either** value1 or value2 of the logical-or function are **true**, then the result is true. If both values are false, then result is false.



### 7.5.5.9 *not-equal*

Result is true if a is not equal to b.





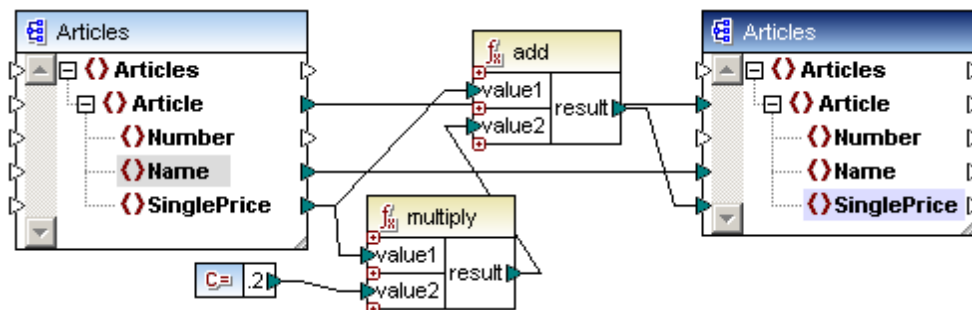
## 7.5.6 core | math functions

Math functions are used to perform basic mathematical functions on data. Note that they cannot be used to perform computations on durations, or datetimes.

input parameters = **value1** | **value2**

output parameter = **result**

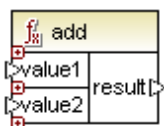
input values are automatically converted to decimal for further processing.



The example shown above, adds 20% sales tax to each of the articles mapped to the target component.

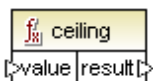
### 7.5.6.1 add

Result is the decimal value of adding **value1** to **value2**.



### 7.5.6.2 ceiling

Result is the smallest integer that is greater than or equal to **value**, i.e. the next highest integer value of the decimal input **value**.

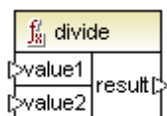


E.g. if the result of a division function is 11.2, then applying the ceiling function to it makes the result 12, i.e. the next highest whole number.



### 7.5.6.3 divide

Result is the decimal value of dividing **value1** by **value2**. The result precision depends on the target language. Use the [round-precision](#) function to define the precision of result.



### 7.5.6.4 floor

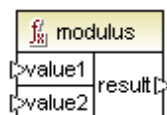
Result is the largest integer that is less than or equal to **value**, i.e. the next lowest integer value of the decimal input **value**.



E.g. if the result of a division function is 11.2, then applying the floor function to it makes the result 11, i.e. the next lowest whole number.

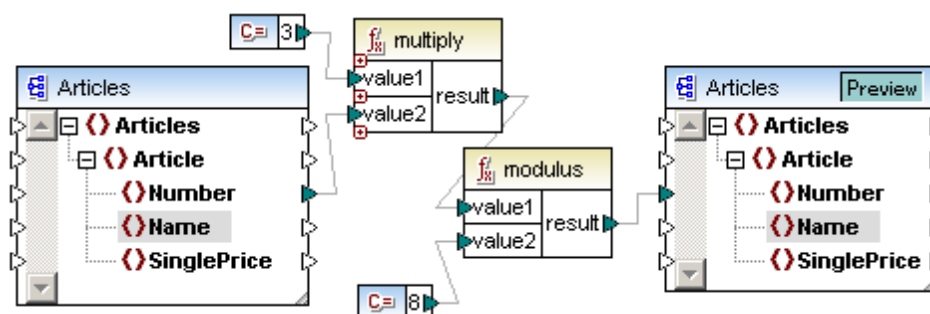
### 7.5.6.5 modulus

Result is the remainder of dividing **value1** by **value2**.



In the mapping below, the numbers have been multiplied by 3 and passed on to value1 of the modulus function. Input values are now 3, 6, 9, and 12.

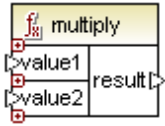
When applying/using modulus 8 as value2, the remainders are 3, 6, 1, and 4.





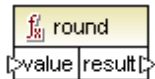
### 7.5.6.6 *multiply*

Result is the decimal value of multiplying **value1** by **value2**.



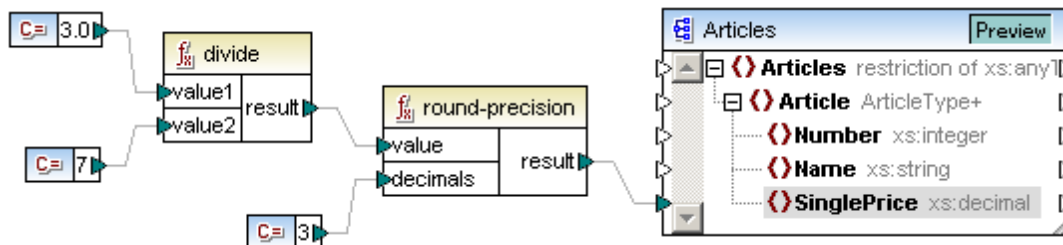
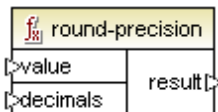
### 7.5.6.7 *round*

Returns the value rounded to the nearest integer. When the value is exactly in between two integers, the "Round Half Towards Positive Infinity" algorithm is used. For example, the value "10.5" gets rounded to "11", and the value "-10.5" gets rounded to "-10".



### 7.5.6.8 *round-precision*

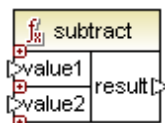
Result is the decimal value of the number rounded to the decimal places defined by "decimals".



In the mapping above, the result is 0.429. For the result to appear correctly in an XML file, make sure to map it to an element of xs:decimal type.

### 7.5.6.9 *subtract*

Result is the decimal value of subtracting **value2** from **value1**.



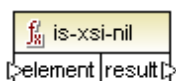


## 7.5.7 core | node functions

The node functions allow you to access nodes, or process nodes in a particular way.

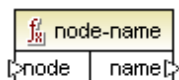
### 7.5.7.1 *is-xsi-nil*

Returns true (`<OrderID>true</OrderID>`) if the element node, of the source component, has the `xsi:nil` attribute set to "true".



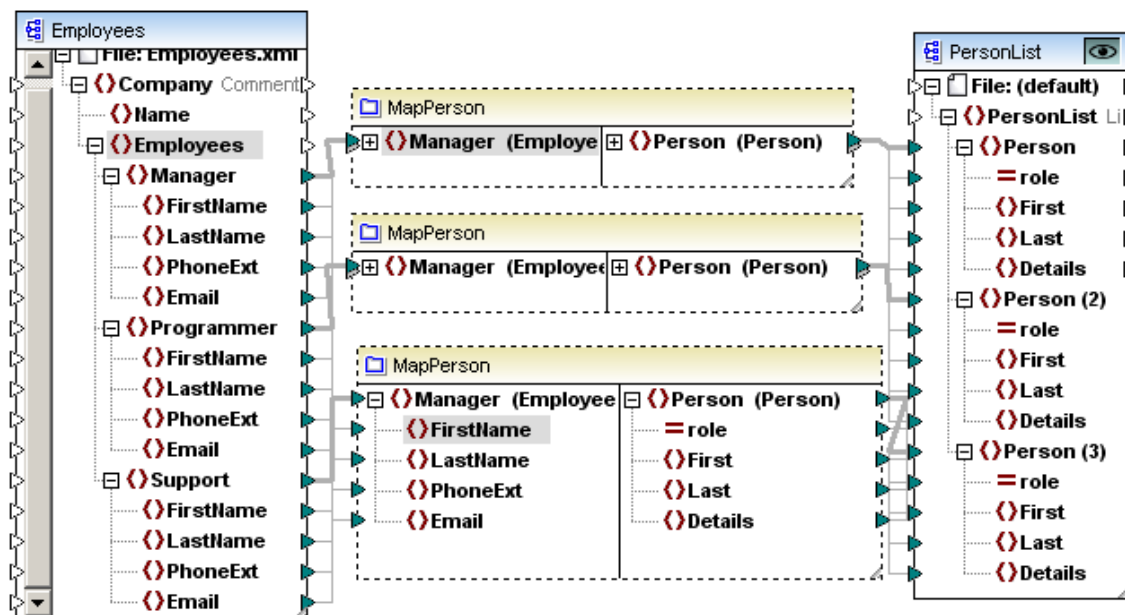
### 7.5.7.2 *node-name*

Returns the qualified name (QName) of the connected node. If the node is an XML **text()** node, an empty QName is returned. This function only works on those nodes that have a name. If XSLT is the target language (which calls `fn:node-name`), the function returns an empty sequence for nodes which have no names.

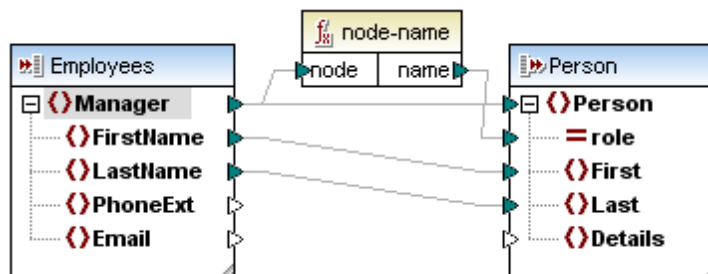


- Getting a name from database tables/fields is not supported.
- XBRL and Excel are not supported.
- Getting a name of File input node is not supported.
- WebService nodes behave like XML nodes except that:
  - node-name from "part" is not supported.
  - node-name from root node ("Output" or "Input") is not supported.





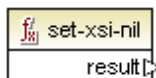
The MapPerson user-defined function uses **node-name** to return the name of the input **node**, and place it in the role attribute. The root node of the Employees.xsd, in the user-defined function, has been defined as "Manager".



Manager gets its data from **outside** the user-defined function, where it can be either: Manager, Programmer, or Support. This is the data that is then passed on to the role attribute in PersonList.

### 7.5.7.3 set-xsi-nil

Sets the target node to xsi:nil.



### 7.5.7.4 static-node-annotation

Returns the string with annotation of the connected node. The input must be: (i) a [source component](#) node, or (ii) an [inline function](#) that is directly connected to a [parameter](#), which in turn is directly connected to a node in the calling mapping.

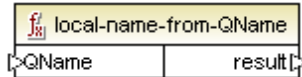






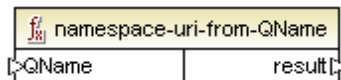
### 7.5.8.2 *local-name-from-QName*

Returns the local name part of the QName.



### 7.5.8.3 *namespace-uri-from-QName*

Returns the namespace URI part of the QName.



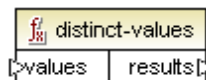
## 7.5.9 core | sequence functions

Sequence functions allow processing of input sequences and grouping of their content. The value/content of the **key** input parameter, mapped to nodes/rows, is used to group the sequence.

- Input parameter **key** is of an arbitrary data type that can be converted to string for **group-adjacent** and **group-by**
- Input parameter **bool** is of type Boolean for **group-starting-with** and **group-ending-with**
- The output **key** is the key of the current group.

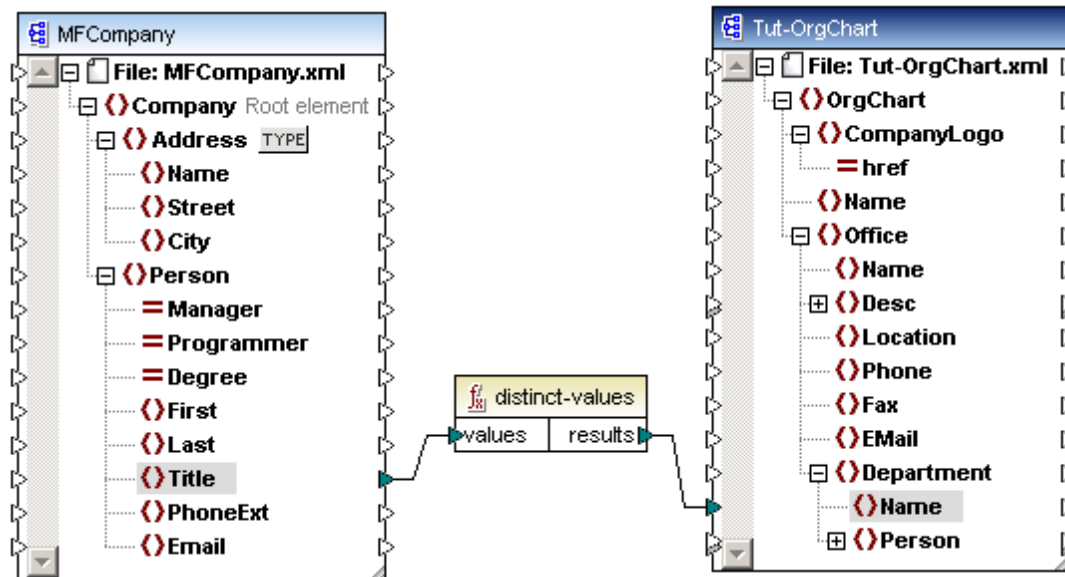
### 7.5.9.1 *distinct-values*

Allows you to remove duplicate values from a sequence and map the unique items to the target component.



In the example below, the content of the source component "Title" items, are scanned and each unique title is mapped to the Department / Name item in the target component.





Note that the sequence of the individual Title items in the source component are retained when mapped to the target component.

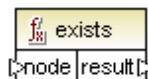
```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <OrgChart xsi:schemaLocation="http://www.xmlspy.com/schemas/orgchart C:/DOCU
3  <Office>
4  <Department>
5      <Name>Office Manager</Name>
6      <Name>Accounts Receivable</Name>
7      <Name>Accounting Manager</Name>
8      <Name>Marketing Manager Europe</Name>
9      <Name>Art Director</Name>
10     <Name>Program Manager</Name>
11     <Name>Software Engineer</Name>
12     <Name>Technical Writer</Name>
13     <Name>IT Manager</Name>
14     <Name>Web Developer</Name>
15     <Name>Support Engineer</Name>
16     <Name>PR & Marketing Manager US</Name>
17 </Department>
18 </Office>
19 </OrgChart>

```

### 7.5.9.2 exists

Returns true if the node exists, else returns false.

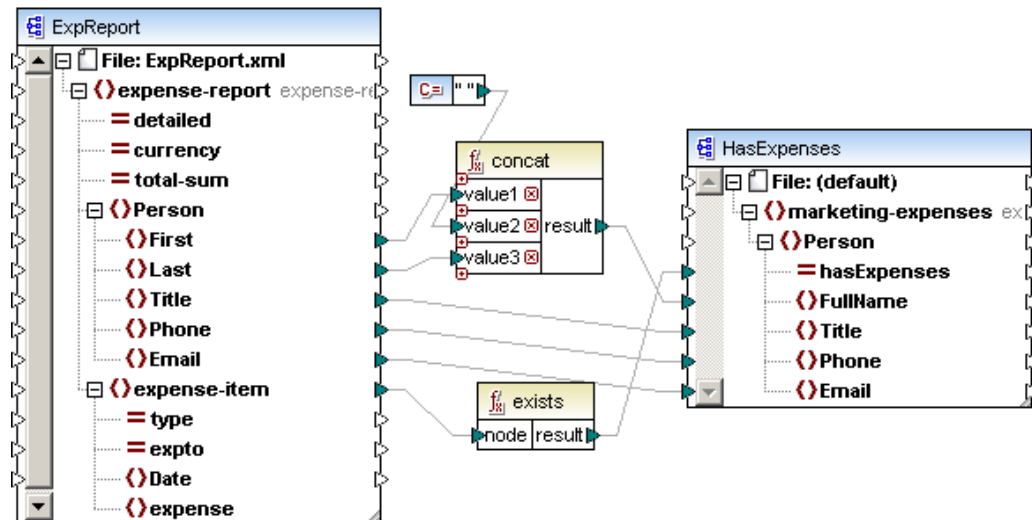


The "**HasMarketingExpenses.mfd**" file in the [.../MapForceExamples](#) folder contains the small example shown below.

If an expense-item exists in the source XML, then the "hasExpenses" attribute is set to "true" in

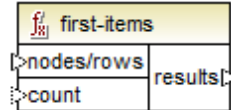


the target XML/Schema file.



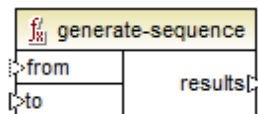
### 7.5.9.3 first-items

Returns the first "X" items of the input sequence, where X is the number supplied by the "count" parameter. E.g. if the value 3 is mapped to the count parameter and a parent node to the nodes/row parameter, then the first three items will be listed in the output.



### 7.5.9.4 generate-sequence

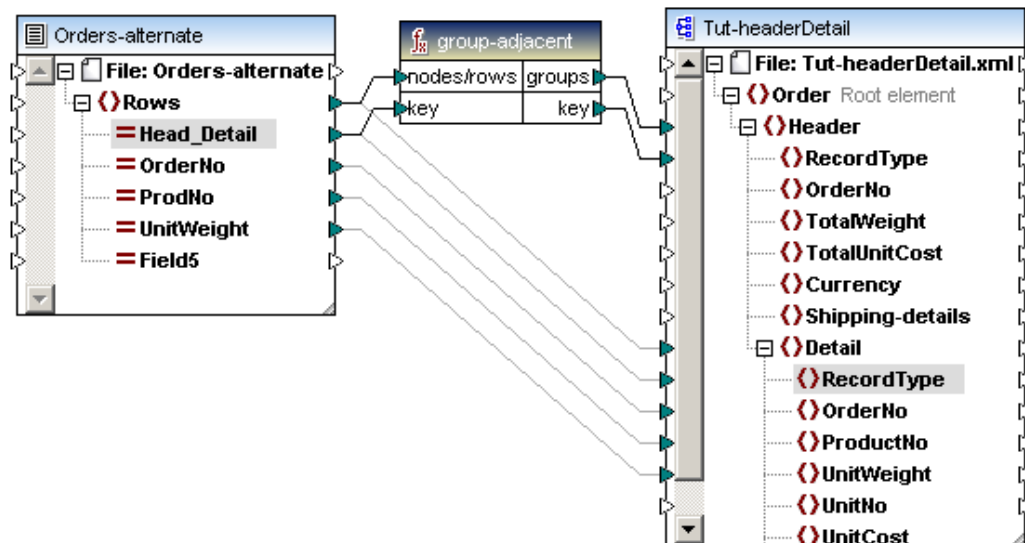
Creates a sequence of integers using the "from" and "to" parameters as the boundaries.



### 7.5.9.5 group-adjacent

Groups the input sequence **nodes/rows** into groups of adjacent items sharing the same **key**. Note that group-adjacent uses the **content** of the node/item as the grouping key.

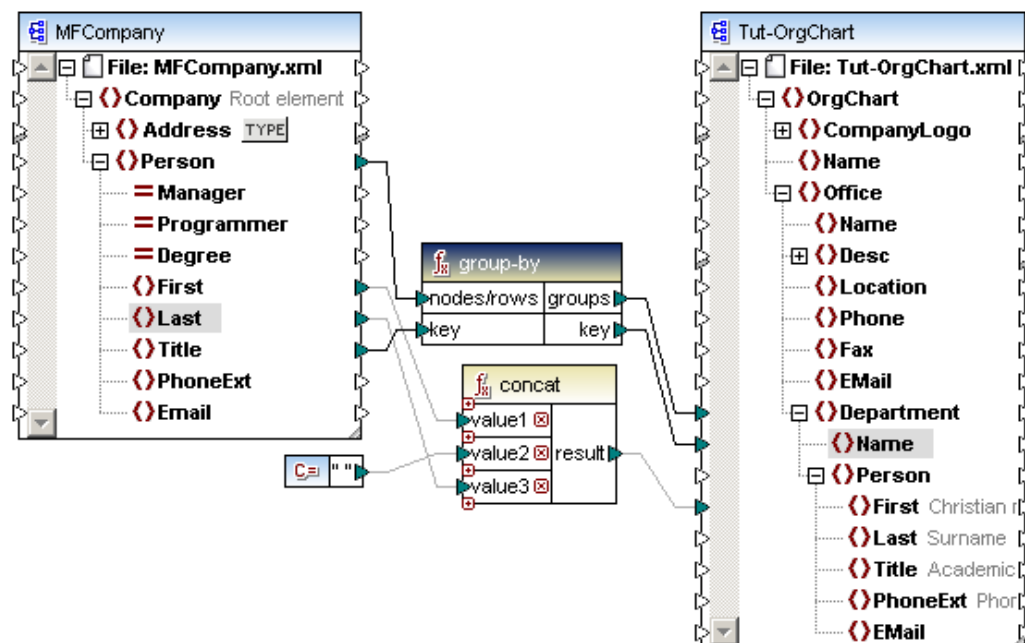




### 7.5.9.6 group-by

Groups the input sequence **nodes/rows** into groups of not necessarily adjacent items sharing the same **key**. Groups are output in the order the key occurs in the input sequence. The example below shows how this works:

- The key that defines the specific groups of the source component is the Title item. This is used to group the persons of the company.
- The group name is placed in the Department/Name item of the target component, while the concatenated person's first and last names are placed in the Person/First child item.



Note that group-by uses the **content** of the node/item as the grouping key. The content of the



Title field is used to group the persons and is mapped to the Department/Name item in the target.

Note also: there is an **implied filter** of the rows from the source document to the target document, which can be seen in the included example. In the target document, each Department item only has those Person items that **match** the grouping **key**, as the group-by component creates the necessary hierarchy on the fly.

Clicking the Output button shows the result of the grouping process.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <OrgChart xsi:schemaLocation="http://www.xmlspy.com/schemas/orgchart C:/DOCU
3  <Office>
4  <Department>
5    <Name>Office Manager</Name>
6    <Person>
7      <First>Vernon Callaby</First>
8      <First>Steve Meier</First>
9    </Person>
10 </Department>
11 <Department>
12   <Name>Accounts Receivable</Name>
13   <Person>
14     <First>Frank Further</First>
15     <First>Theo Bone</First>
16   </Person>
17 </Department>

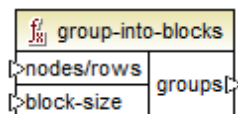
```

#### 7.5.9.7 *group-ending-with*

This function groups the input sequence **nodes/rows** into groups, ending a new group whenever **bool** is true.

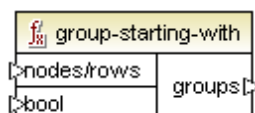
#### 7.5.9.8 *group-into-blocks*

Groups the input sequence **nodes/rows** into blocks of the same size defined by the number supplied by the **block-size** parameter.



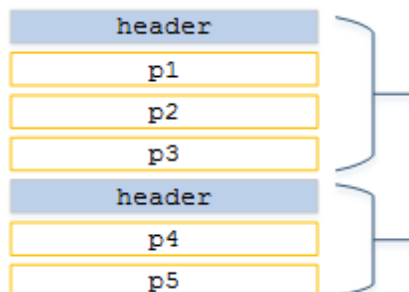
#### 7.5.9.9 *group-starting-with*

This function groups the input sequence **nodes/rows** into groups, starting a new group when **bool** is true.

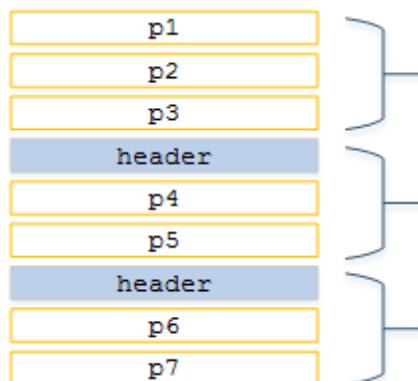




The following example illustrates a sequence of nodes where **bool** returns `true` whenever the node "header" is encountered. Applying the **group-starting-with** function on this sequence of nodes results in two groups, as shown below.

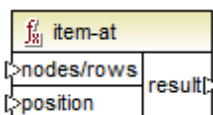


Note that the first node in the sequence starts a new group regardless of the value of **bool**. In other words, a sequence such as the one below would create three groups.



#### 7.5.9.10 *item-at*

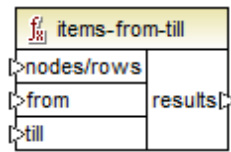
Returns the **nodes/rows** at the position supplied by the **position** parameter. The first item is at position "1".



#### 7.5.9.11 *items-from-till*

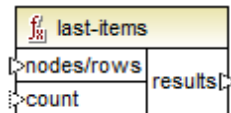
Returns a sequence of **nodes/rows** using the "from" and "till" parameters as the boundaries. The first item is at position "1".





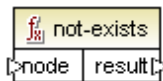
### 7.5.9.12 last-items

Returns the last "X" **nodes/rows** of the sequence where X is the number supplied by the "count" parameter. The first item is at position "1".



### 7.5.9.13 not-exists

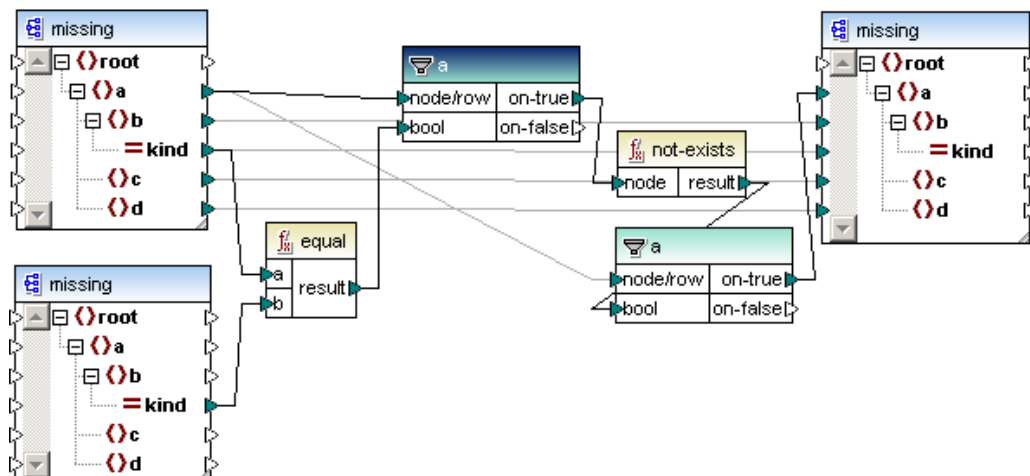
Returns false if the node exists, else returns true.



The example below shows how you can use the not-exists function to map nodes that do not exist in one of a pair of source files.

What this mapping does:

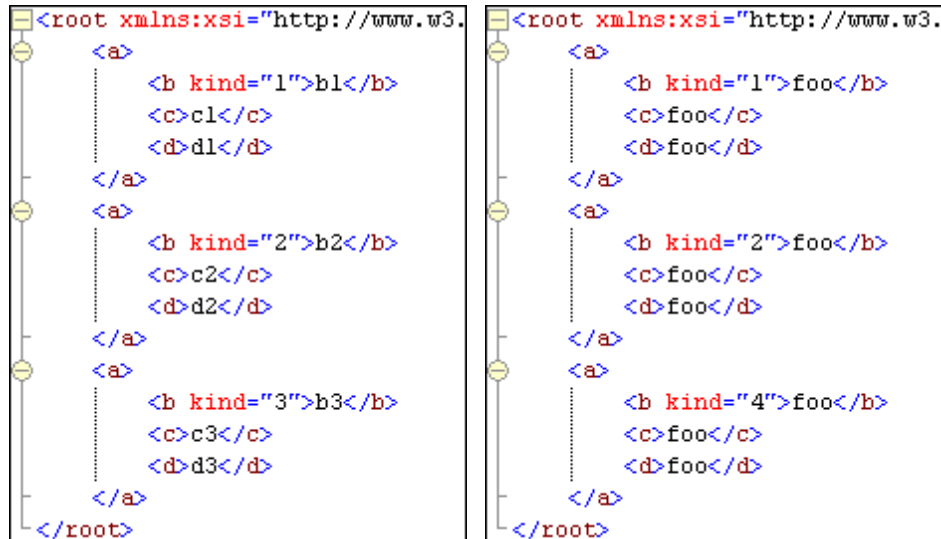
- Compare the nodes of two source XML files
- Filter out the nodes of the first source XML file, that do not exist in the second source XML file
- Map only the missing nodes, and their content, to the target file.



The two XML instance files are shown below, the differences between them are:

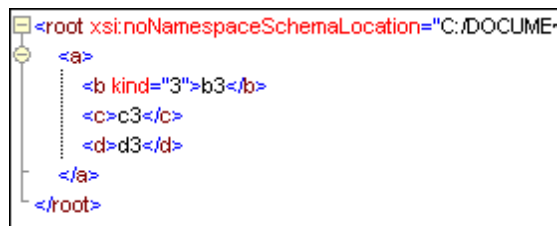


- **a.xml** (left) contains the node `<b kind="3">`, which is missing from b.xml.
- **b.xml** (right) contains the node `<b kind="4">` which is missing from a.xml.



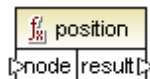
- The equal function compares the kind attribute of both XML files and passes the result to the filter.
- A not-exists function is placed after the initial filter, to select the missing nodes of each of the source files.
- The second filter is used to pass on the missing node and other data only from the a.xml file to the target.

The mapping result is that the node missing from b.xml, `<b kind="3">`, is passed on to the target component.



### 7.5.9.14 position

Returns the position of a node inside its containing sequence.

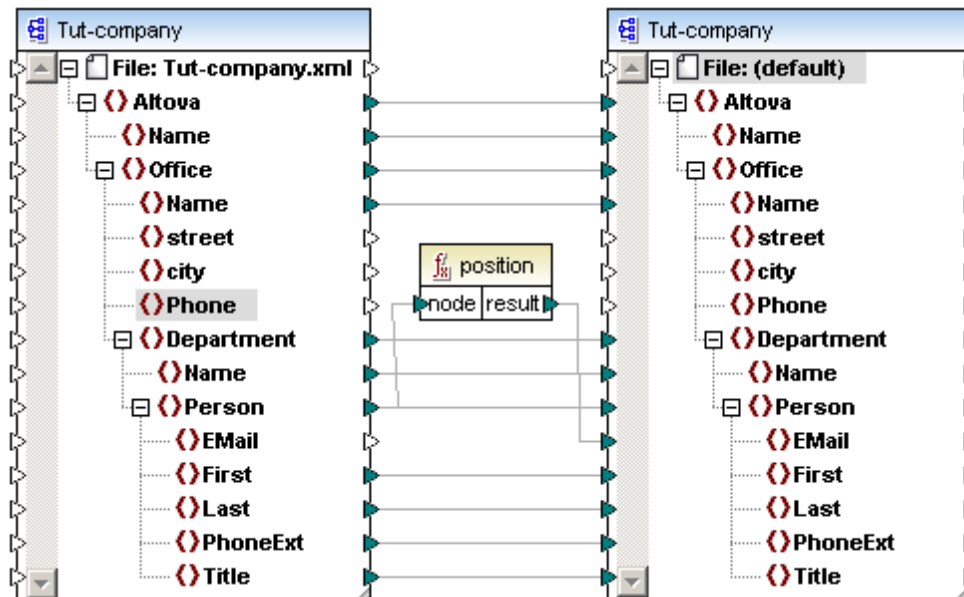


The position function allows you to determine the position of a specific node in a sequence, or use a specific position to filter out items based on that position.

The context item is defined by the item connected to the "node" parameter of the position function, Person, in the example below.



The simple mapping below adds a position number to each Person of each Department.



The position number is reset for each Department in the Office.

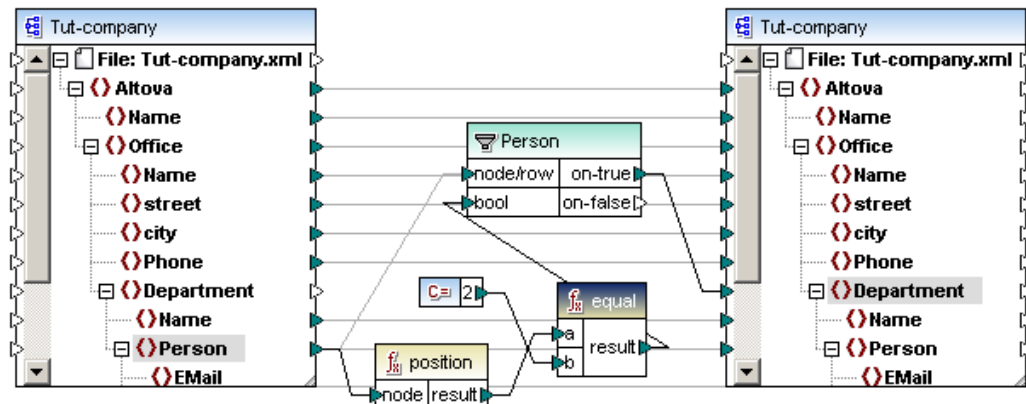
```
<Office>
  <Name>Microtech, Inc.</Name>
  <Department>
    <Name>Admin</Name>
    <Person>
      <EMail>1</EMail>
      <First>Albert</First>
      <Last>Aldrich</Last>
      <PhoneExt>582</PhoneExt>
      <Title>Manager</Title>
    </Person>
    <Person>
      <EMail>2</EMail>
      <First>Bert</First>
      <Last>Bander</Last>
      <PhoneExt>471</PhoneExt>
      <Title>Accounts Receivable</Title>
    </Person>
  </Department>
</Office>
```

Using the position function to filter out specific nodes

Using the position function in conjunction with a filter allows you to map only those specific nodes that have a certain position in the source component.

The filter "node/row" parameter and the position "node" must be connected to the same item of the source component, to filter out a specific position of that sequence.





What this mapping does is to output:

- The second Person in each Department
- of each Office in Altova.

```
<Office>
  <Name>Microtech, Inc.</Name>
  <Department>
    <Name>Admin</Name>
    <Person>
      <Email>b.bander@microtech.com</Email>
      <First>Bert</First>
      <Last>Bander</Last>
      <Title>Accounts Receivable</Title>
    </Person>
  </Department>
  <Department>
    <Name>Sales and Marketing</Name>
    <Person>
      <Email>e.ellas@microtech.com</Email>
      <First>Eve</First>
      <Last>Elas</Last>
      <Title>Art Director</Title>
    </Person>
  </Department>
  <Department>
    <Name>Manufacturing</Name>
    <Person>
      <Email>g.gundall@microtech.com</Email>
```

Finding the position of items in a filtered sequence:

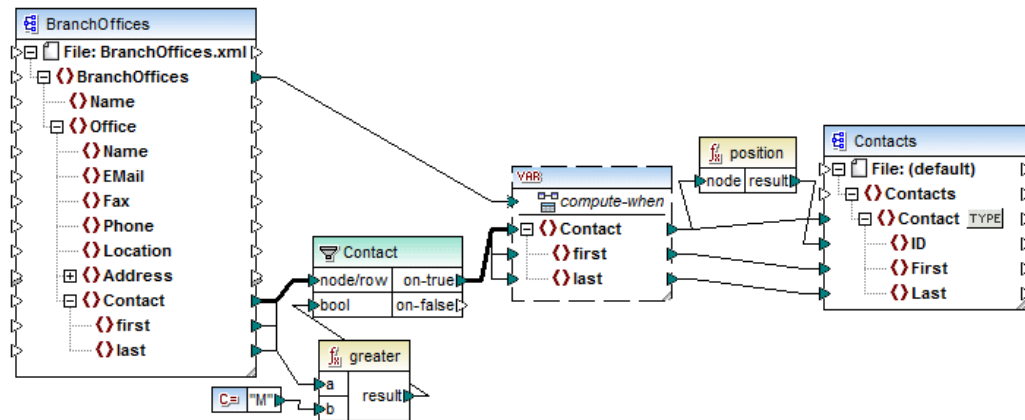
As the filter component is not a sequence function, it cannot be used directly in conjunction with the position function to find the position of filtered items. To do this you have to use the "[Variable](#)" component.

The results of a Variable component are always sequences, i.e. a delimited list of values, which can also be used to create sequences.

- The variable component is used to collect the filtered contacts where the last name starts with a letter higher than "M".
- The contacts are then passed on (from the variable) to the target component

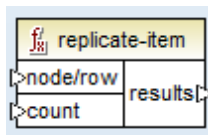


- The position function then numbers these contacts sequentially



### 7.5.9.15 replicate-item

Repeats every item in the input sequence the number of times specified in the `count` argument. If you connect a single item to the `node/row` argument, the function returns `N` items, where `N` is the value of the `count` argument. If you connect a sequence of items to the `node/row` argument, the function repeats each individual item in the sequence `count` times, processing one item at a time. For example, if `count` is 2, then the sequence (1,2,3) produces (1,1,2,2,3,3).



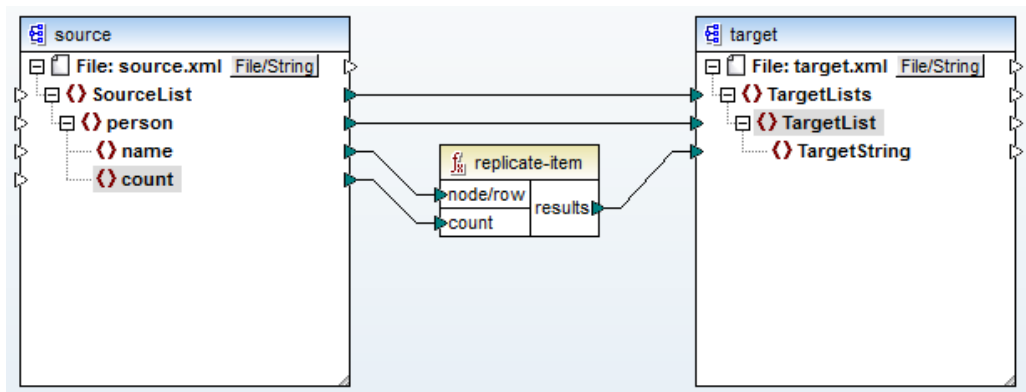
Note that you can supply a different `count` value for each item. For example, let's assume that you have a source XML file with the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<SourceList xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="source.xsd">
  <person>
    <name>Michelle</name>
    <count>2</count>
  </person>
  <person>
    <name>Ted</name>
    <count>4</count>
  </person>
  <person>
    <name>Ann</name>
    <count>3</count>
  </person>
</SourceList>
```

With the help of the **replicate-item** function, you can repeat each person name a different number of times in a target component. To achieve this, connect the `<count>` node of each



person to the `count` input of the **replicate-item** function:

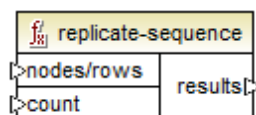


The output is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<TargetLists xsi:noNamespaceSchemaLocation="target.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <TargetList>
    <TargetString>Michelle</TargetString>
    <TargetString>Michelle</TargetString>
  </TargetList>
  <TargetList>
    <TargetString>Ted</TargetString>
    <TargetString>Ted</TargetString>
    <TargetString>Ted</TargetString>
    <TargetString>Ted</TargetString>
  </TargetList>
  <TargetList>
    <TargetString>Ann</TargetString>
    <TargetString>Ann</TargetString>
    <TargetString>Ann</TargetString>
  </TargetList>
</TargetLists>
```

### 7.5.9.16 replicate-sequence

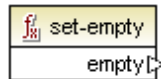
Repeats all items in the input sequence the number of times specified in the `count` argument. For example, if `count` is 2, then the sequence (1,2,3) produces (1,2,3,1,2,3).





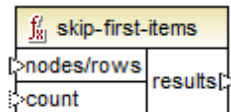
### 7.5.9.17 *set-empty*

Returns an empty sequence.



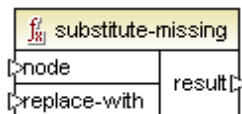
### 7.5.9.18 *skip-first-items*

Skips the first "X" items/nodes of the input sequence, where X is the number supplied by the "count" parameter, and returns the rest of the sequence.



### 7.5.9.19 *substitute-missing*

This function is a convenient combination of exists and a suitable if-else condition. Used to map the current field content if the node exists in the XML source file, otherwise use the item mapped to the "replace-with" parameter.

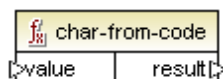


## 7.5.10 core | string functions

The string functions allow you to use the most common string functions to manipulate many types of source data to: extract portions, test for substrings, or retrieve information on strings.

### 7.5.10.1 *char-from-code*

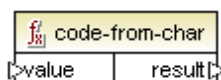
Result is the character representation of the decimal Unicode value of **value**.





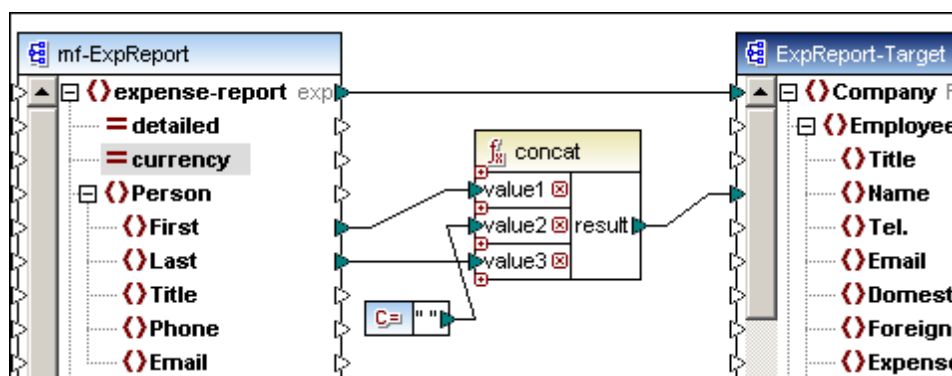
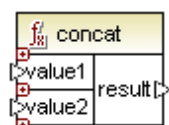
### 7.5.10.2 *code-from-char*

Result is the decimal Unicode value of the first character of **value**.



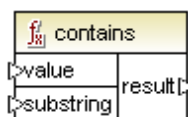
### 7.5.10.3 *concat*

Concatenates (appends) two or more values into a single result string. All input values are automatically converted to type string.



### 7.5.10.4 *contains*

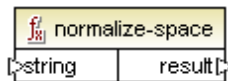
Result is true if data supplied to the value parameter contains the string supplied by the substring parameter.



### 7.5.10.5 *normalize-space*

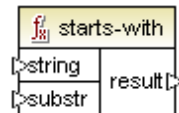
Result is the normalized input string, i.e. leading and trailing spaces are removed, then each sequence of multiple consecutive whitespace characters are replaced by a single whitespace character. The Unicode character for "space" is (U+0020).





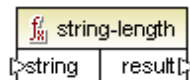
### 7.5.10.6 starts-with

Result is true if the input string "string" starts with **substr**, else false.



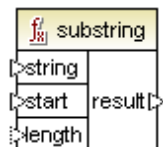
### 7.5.10.7 string-length

Result is the number of characters supplied by the **string** parameter.



### 7.5.10.8 substring

Result is the substring (string fragment) of the "string" parameter where "start" defines the position of the start character, and "length" the length of the substring.

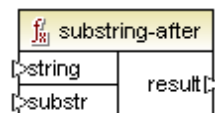


If the length parameter is not specified, the result is a fragment starting at the start position and ending at the end position of the string. Indices start counting at 1.

E.g. substring("56789",2,3) results in 678.

### 7.5.10.9 substring-after

Result is the remainder of the "**string**" parameter, where the first occurrence of the **substr** parameter defines the start characters; the remainder of the string is the result of the function. An empty string is the result, if **substr** does not occur in **string**.

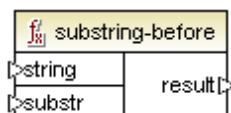


E.g. substring-after("2009/01/04","/") results in the substring 01/04. substr in this case is the first "/" character.



### 7.5.10.10 substring-before

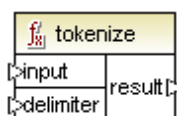
Result is the string fragment of the "**string**" parameter, up to the first occurrence of the **substr** characters. An empty string is the result, if **substr** does not occur in **string**.



E.g. substring-before ("2009/01/04", "/") results in the substring 2009. substr in this case is the first "/" character.

### 7.5.10.11 tokenize

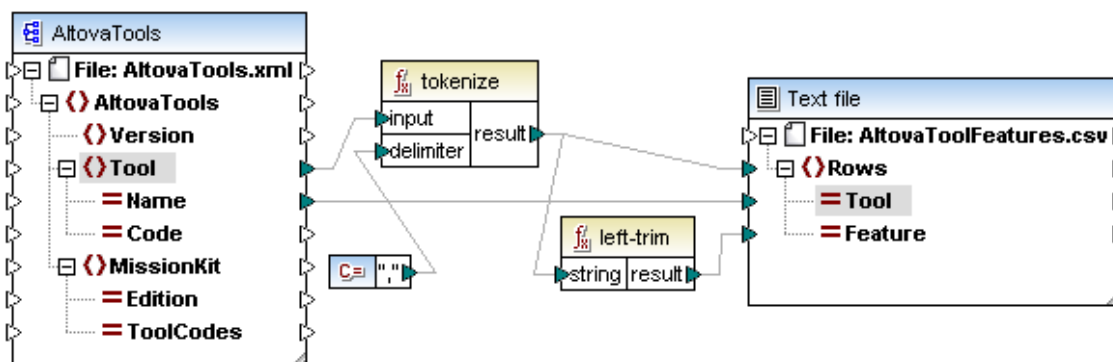
Result is the input string split into a sequence of chunks/sections defined by the delimiter parameter. The result can then be passed on for further processing.



E.g. Input string is A,B,C and delimiter is "," - then result is A B C.

Example

The **tokenizeString1.mfd** file available in the ...\\MapForceExamples folder shows how the tokenize function is used.



The XML source file is shown below. The **Tool** element has two attributes: Name and Code, with the Tool element data consisting of comma delimited text.



AltovaTools

xmlns:xsi

http://www.w3.org/2001/XMLSchema-instance

xsi:noName...

AltovaTools.xsd

Version

2010

Tool (9)

	Name	Code	Text
1	XMLSpy	XS	XML editor, XSLT editor, XSLT debugger, XQuery editor, XQuery debugger,
2	MapForce	MF	Data integration, XML mapping, database mapping, text conversion, EDI tran
3	StyleVision	SV	Stylesheet designer, electronic forms, XSLT design, XSL:FO design, databa
4	UModel	UM	UML modeling tool, code generation, reverse engineering, UML, BPMN, Systm
5	DatabaseSpy	DS	Multi-database tool, SQL auto-completion, graphical database design, table b
6	DiffDog	DD	Diff / merge tool, compare files, sync directories, compare XML, compare O
7	SchemaAgent	SA	XML Schema management tool, IIR management, XSLT management, WSDL
8	SemanticWorks	SW	Semantic Web tool, RDF editor, OWL editor, RDF/XML and N-Triples generati
9	Authentic	AU	XML authoring tool, database editor, XML publishing tool, e-Forms editor

MissionKit (4)

### What the mapping does:

- The tokenize function receives data from the **Tool** element/item and uses the comma "," delimiter to split that data into separate chunks. I.e. the first chunk "XML editor".
- As the result parameter is mapped to the **Rows** item in the target component, one **row** is generated for each chunk.
- The result parameter is also mapped to the **left-trim** function which removes the leading white space of each chunk.
- The result of the left-trim parameter (each chunk) is mapped to the **Feature** item of the target component.
- The target component output file has been defined as a CSV file (AltovaToolFeatures.csv) with the field delimiter being a semicolon (double click component to see settings).

### Result of the mapping:

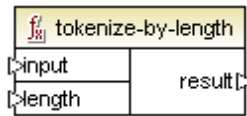
- For each Tool element of the source file
- The (Tool) Name is mapped to the Tool item in the target component
- Each chunk of the tokenized Tool content is appended to the (Tool Name) Feature item
- E.g. The first tool, XMLSpy, gets the first Feature chunk "XML editor"
- This is repeated for all chunks of the current Tool and then for all Tools.
- Clicking the Output tab delivers the result shown below.

1	Tool;Feature
2	XMLSpy;XML editor
3	XMLSpy;XSLT editor
4	XMLSpy;XSLT debugger
5	XMLSpy;XQuery editor
6	XMLSpy;XQuery debugger
7	XMLSpy;XML Schema / DTD editor
8	XMLSpy;WSDL editor
9	XMLSpy;SOAP debugger
10	MapForce;Data integration
11	MapForce;XML mapping
12	MapForce;database mapping



### 7.5.10.12 *tokenize-by-length*

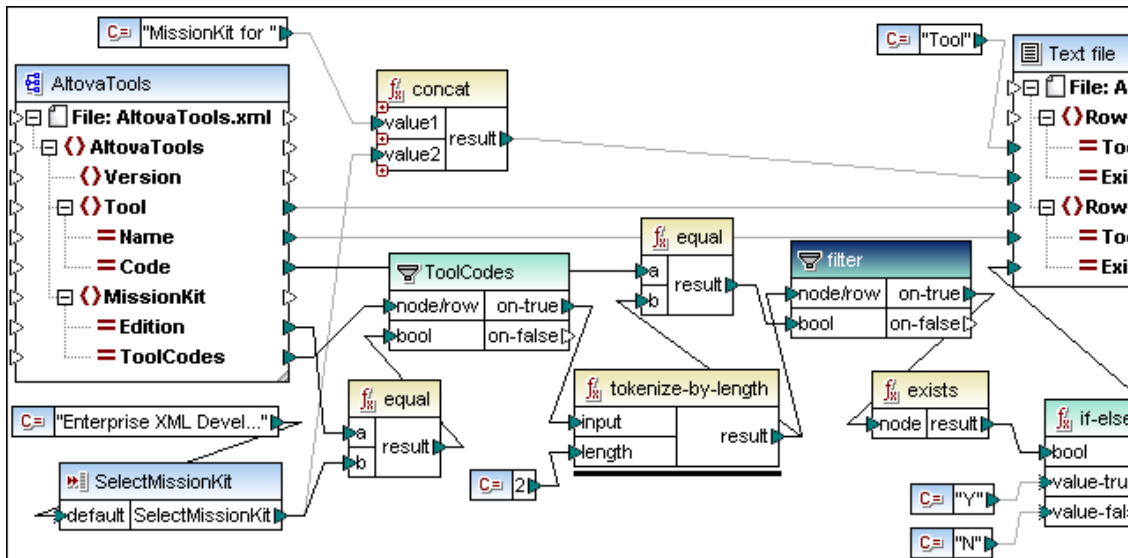
Result is the input string split into a sequence of chunks/sections defined by the length parameter. The result can then be passed on for further processing.



E.g. Input string is ABCDEF and length is "2" - then result is AB CD EF.

Example

The **tokenizeString2.mfd** file available in the ...\\MapForceExamples folder shows how the **tokenize-by-length** function is used.



The XML source file is shown below, and is the same as the one used in the previous example. The **MissionKit** element also has two attributes: **Edition** and **ToolCodes**, but no MissionKit element content.



Tool (9)			
	Name	Code	Text
1	XMLSpy	XS	XML editor, XSLT editor, XSLT debugger, XQuery editor, XQuery debugger, XML S
2	MapForce	MF	Data integration, XML mapping, database mapping, text conversion, EDI translator,
3	StyleVision	SV	Stylesheet designer, electronic forms, XSLT design, XSL:FO design, database rep
4	UModel	UM	UML modeling tool, code generation, reverse engineering, UML, BPMN, SysML, pro
5	DatabaseSpy	DS	Multi-database tool, SQL auto-completion, graphical database design, table brows
6	DiffDog	DD	Diff / merge tool, compare files, sync directories, compare XML, compare OOXML,
7	SchemaAgent	SA	XML Schema management tool, IIR management, XSLT management, WSDL manag
8	SemanticWorks	SW	Semantic Web tool, RDF editor, OWL editor, RDF/XML and N-Triples generation and
9	Authentic	AU	XML authoring tool, database editor, XML publishing tool, e-Forms editor

MissionKit (4)		
	Edition	ToolCodes
1	Enterprise Software Architects	XSMFSVUMDSDDSASW
2	Professional Software Architects	XSMFSVUMDS
3	Enterprise XML Developers	XSMFSVDDDSASW
4	Professional XML Developers	XSMFSV

### Aim of the mapping:

To generate a list showing which Altova tools are part of the respective MissionKit editions.

### How the mapping works:

- The SelectMissionKit **Input** component receives its default input from a constant component, in this case "Enterprise XML Developers".
- The **equal** function compares the input value with the "**Edition**" value and passes on the result to the **bool** parameter of the ToolCodes filter.
- The **node/row** input of the ToolCodes filter is supplied by the **ToolCodes** item of the source file. The value for the Enterprise XML Developers edition is: XSMFSVDDDSASW.
- The XSMFSVDDDSASW value is passed to the **on-true** parameter, and further to the **input** parameter of the **tokenize-by-length** function.

### What the tokenize-by-length function does:

- The ToolCodes **input** value XSMFSVDDDSASW, is split into multiple chunks of two characters each, defined by **length** parameter, which is 2, thus giving 6 chunks.
- Each chunk (placed in the **b** parameter) of the equal function, is compared to the 2 character **Code** value of the source file (of which there are 9 entries/items in total).
- The result of the comparison (true/false) is passed on to the **bool** parameter of the filter.
- Note that **all** chunks, of the tokenize-by-length function, are passed on to the **node/row** parameter of the filter.
- The **exists** functions now checks for existing/non-existing nodes passed on to it by the **on-true** parameter of the filter component.

Existing nodes are those where there **is** a match between the **ToolCodes** chunk and the Code value.

Non-existing nodes are where there was **no ToolCodes** chunk to match a Code value.

- The bool results of the **exists** function are passed on to the **if-else** function which passes on a Y to the target if the node exists, or a N, if the node does not exist.

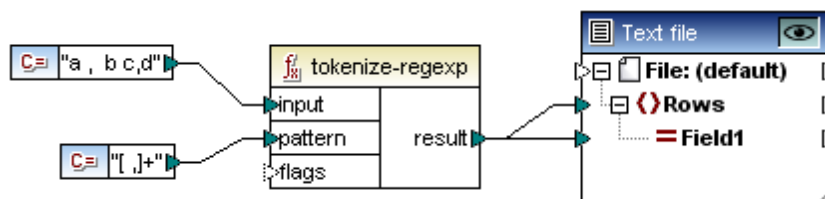
Result of the mapping:



1	Tool;MissionKit for Enterprise XML Developers
2	XMLSpy;Y
3	MapForce;Y
4	StyleVision;Y
5	UModel;N
6	DatabaseSpy;N
7	DiffDog;Y
8	SchemaAgent;Y
9	SemanticWorks;Y
10	Authentic;N
11	

### 7.5.10.13 tokenize-regexp

Result is the input string split into a sequence of strings, where the supplied regular expression **pattern** match defines the separator. The separator strings are not output by the result parameter. Optional flags may also be used.



In the example shown above:

**input** string is a succession of characters separated by spaces and/or commas, i.e. a , b c,d

The regex **pattern** defines a character class ["space"comma"] - of which one and only one character will be matched in a character class, i.e. either space or comma.

The **+** quantifier specifies "one or more" occurrences of the character class/string.

result string is:

1	a
2	b
3	c
4	d
5	

Please note that there are slight differences in regular expression syntax between the various languages. Tokenize-regexp in C++ is only available in Visual Studio 2008 SP1 and later.

For more information on regular expressions, see [Regular expressions](#).

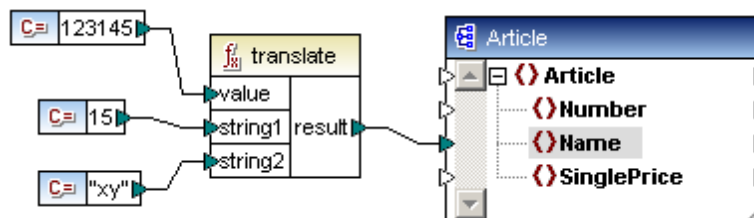


### 7.5.10.14 *translate*

The characters of **string1** (search string) are replaced by the characters at the same position in **string2** (**replace string**), in the input string "**value**".



When there are no corresponding characters in string2, the character is removed.



E.g.

input string is 123145  
           (search) string1 is 15  
           (replace) string2 is xy

So:

each 1 is replaced by **x** in the input string value  
 each 5 is replaced by **y** in the input string value

Result string is **x23x4y**

If string2 is empty (fewer characters than string1) then the character is removed.

E.g.2

input string aabaacbca  
           string1 is "a"  
           string2 is "" (empty string)

result string is "bcbc"

E.g.3

input string aabaacbca  
           string1 is "ac"  
           string2 is "ca"

result string is "ccbccabac"

## 7.5.11 xpath2 | accessors

XPath2 functions are available when either the XSLT2 or XQuery languages are selected.



### 7.5.11.1 *base-uri*

The `base-uri` function takes a node argument as input, and returns the URI of the XML resource containing the node. The output is of type `xs:string`. MapForce returns an error if no input node is supplied.

### 7.5.11.2 *node-name*

The `node-name` function takes a node as its input argument and returns its QName. When the QName is represented as a string, it takes the form of `prefix:localname` if the node has a prefix, or `localname` if the node has no prefix. To obtain the namespace URI of a node, use the `namespace-uri-from-QName` function (in the library of QName-related functions).

### 7.5.11.3 *string*

The `string` function works like the `xs:string` constructor: it converts its argument to `xs:string`.

When the input argument is a value of an atomic type (for example `xs:decimal`), this atomic value is converted to a value of `xs:string` type. If the input argument is a node, the string value of the node is extracted. (The string value of a node is a concatenation of the values of the node's descendant nodes.)

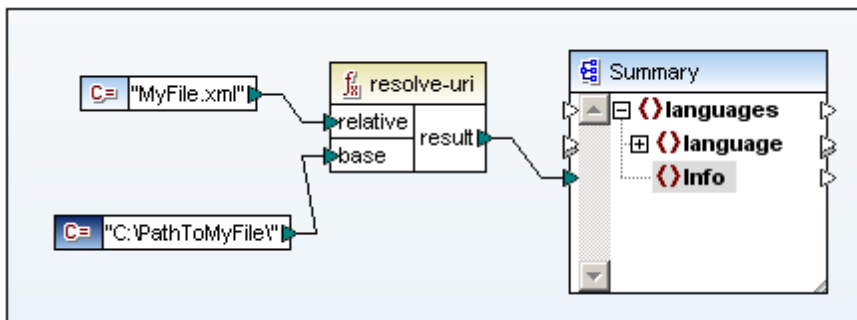
## 7.5.12 xpath2 | anyURI functions

XPath2 functions are available when either the XSLT2 or XQuery languages are selected.

### 7.5.12.1 *resolve-uri*

The `resolve-uri` function takes a URI as its first argument (datatype `xs:string`) and resolves it against the URI in the second argument (datatype `xs:string`).

The result (datatype `xs:string`) is a combined URI. In this way a relative URI (the first argument) can be converted to an absolute URI by resolving it against a base URI.



In the screenshot above, the first argument provides the relative URI, the second argument the



base URI. The resolved URI will be a concatenation of base URI and relative URI, so c:  
 \PathtoMyFile\MyFile.xml.

**Note:** Both arguments are of datatype `xs:string` and the process of combining is done by treating both inputs as strings. So there is no way of checking whether the resources identified by these URIs actually exist. MapForce returns an error if the second argument is not supplied.

### 7.5.13 xpath2 | boolean functions

XPath2 functions are available when either the XSLT2 or XQuery languages are selected. The Boolean functions `true` and `false` take no argument and return the boolean constant values, `true` and `false`, respectively. They can be used where a constant boolean value is required.

#### 7.5.13.1 false

Returns the Boolean value "false".

#### 7.5.13.2 true

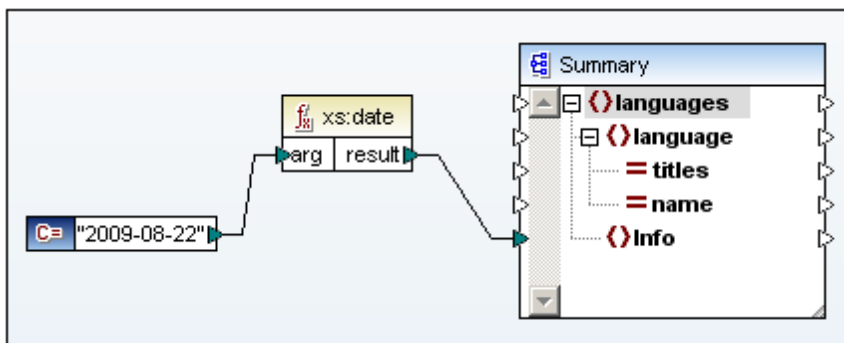
Returns the Boolean value "true".

### 7.5.14 xpath2 | constructors

XPath2 functions are available when either the XSLT2 or XQuery languages are selected.

The functions in the Constructors part of the XPath 2.0 functions library construct specific datatypes from the input text. Typically, the lexical format of the input text must be that expected of the datatype to be constructed. Otherwise, the transformation will not be successful.

For example, if you wish to construct an `xs:date` datatype, use the `xs:date` constructor function. The input text must have the lexical format of the `xs:date` datatype, which is: `YYYY-MM-DD` (screenshot below).



In the screenshot above, a string constant (2009-08-22) has been used to provide the input argument of the function. The input could also have been obtained from a node in the source



document.

The `xs:date` function returns the input text (2009-08-22), which is of `xs:string` datatype (specified in the *Constant* component), as output of `xs:date` datatype.

When you mouseover the input argument in a function box, the expected datatype of the argument is displayed in a popup.

## 7.5.15 xpath2 | context functions

XPath2 functions are available when either the XSLT2 or XQuery languages are selected.

The Context functions library contains functions that provide the current date and time, the default collation used by the processor, and the size of the current sequence and the position of the current node.

### 7.5.15.1 *current-date*

Returns the current date (`xs:date`) from the system clock.

### 7.5.15.2 *current-dateTime*

Returns the current date and time (`xs:dateTime`) from the system clock.

### 7.5.15.3 *current-time*

Returns the current time (`xs:time`) from the system clock.

### 7.5.15.4 *default-collation*

The `default-collation` function takes no argument and returns the default collation, that is, the collation that is used when no collation is specified for a function where one can be specified.

The Altova XSLT 2.0 Engine supports the Unicode codepoint collation only. Comparisons, including for the `fn:max` and `fn:min` functions, are based on this collation.

### 7.5.15.5 *implicit-timezone*

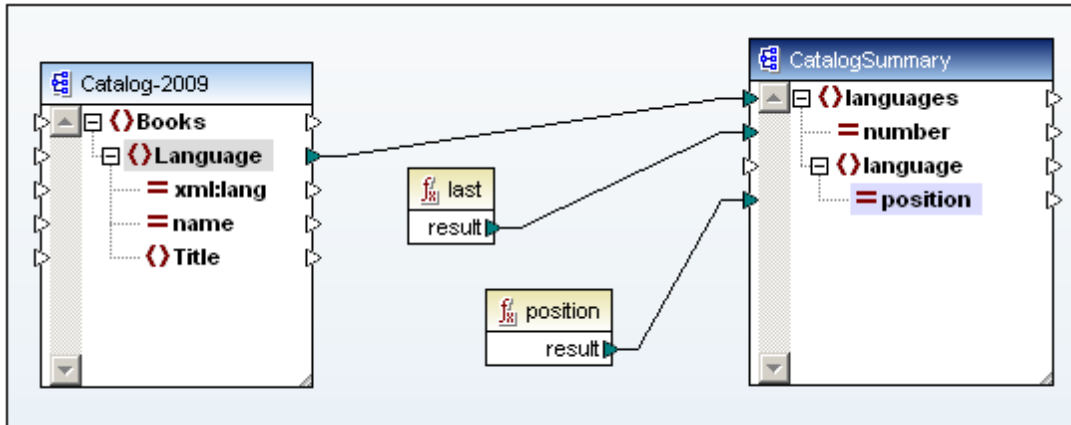
Returns the value of the "implicit timezone" property from the evaluation context.

### 7.5.15.6 *last*

The `last` and `position` functions take no argument. The `last` function returns the position of the last node in the context nodeset. The `position` function returns the position of the current node in the nodeset being processed.



The context nodeset at the nodes where the functions are directed, is the nodeset to which the functions will apply. In the screenshot below, the nodeset of `Language` elements is the context nodeset for the `last` and `position` functions.



In the example above, the `last` function returns the position of the last node of the context nodeset (the nodeset of `Language` elements) as the value of the `number` attribute. This value is also the size of the nodeset since it indicates the number of nodes in the nodeset.

The `position` function returns the position of the `Language` node being currently processed. For each `Language` element node, its position within the nodeset of `Language` elements is output to the `language/@position` attribute node.

We would advise you to use the **position** and **count** functions from the **core** library.

## 7.5.16 xpath2 | durations, date and time functions

XPath2 functions are available when either the XSLT2 or XQuery languages are selected.

The XPath 2 duration and date and time functions enable you to adjust dates and times for the timezone, extract particular components from date-time data, and subtract one date-time unit from another.

### The 'Adjust-to-Timezone' functions

Each of these related functions takes a date, time, or dateTime as the first argument and adjusts the input by adding, removing, or modifying the timezone component depending on the value of the second argument.

The following situations are possible when the first argument contains no timezone (for example, the date 2009-01 or the time 14:00:00).

- Timezone argument (the second argument of the function) is present: The result will contain the timezone specified in the second argument. The timezone in the second argument is added.
- Timezone argument (the second argument of the function) is absent: The result will contain the implicit timezone, which is the system's timezone. The system's timezone is added.
- Timezone argument (the second argument of the function) is empty: The result will



contain no timezone.

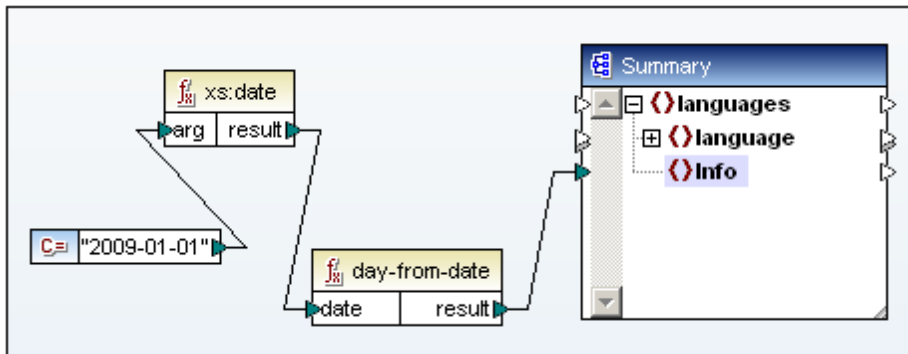
The following situations are possible when the first argument contains a timezone (for example, the date 2009-01-01+01:00 or the time 14:00:00+01:00).

- Timezone argument (the second argument of the function) is present: The result will contain the timezone specified in the second argument. The original timezone is replaced by the timezone in the second argument.
- Timezone argument (the second argument of the function) is absent: The result will contain the implicit timezone, which is the system's timezone. The original timezone is replaced by the system's timezone.
- Timezone argument (the second argument of the function) is empty: The result will contain no timezone.

### The 'From' functions

Each of the 'From' functions extracts a particular component from: (i) date or time data, and (ii) duration data. The results are of the xs:decimal datatype.

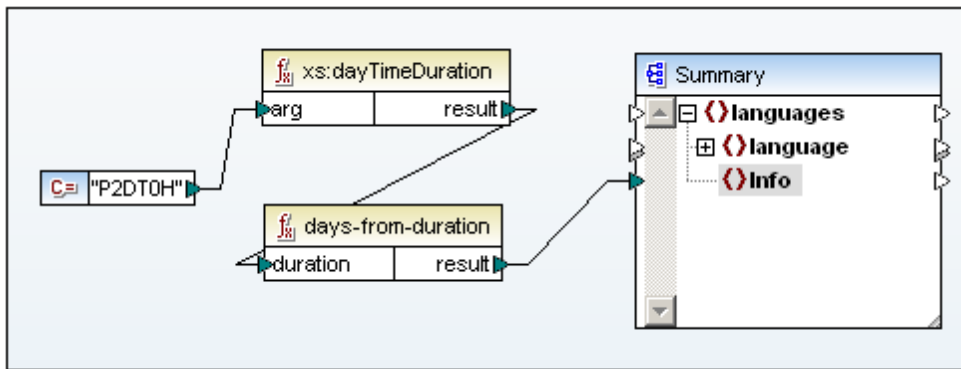
As an example of extracting a component from date or time data, consider the `day-from-date` function (screenshot below).



The input argument is a date (2009-01-01) of type `xs:date`. The `day-from-date` function extracts the day component of the date (1) as an `xs:decimal` datatype.

Extraction of time components from durations requires that the duration be specified either as `xs:yearMonthDuration` (for extracting years and months) or `xs:dayTimeDuration` (for extracting days, hours, minutes, and seconds). The result will be of type `xs:decimal`. The screenshot below shows a `dayTimeDuration` of `P2DT0H` being input to the `days-from-duration` function. The result is the `xs:decimal` 2.

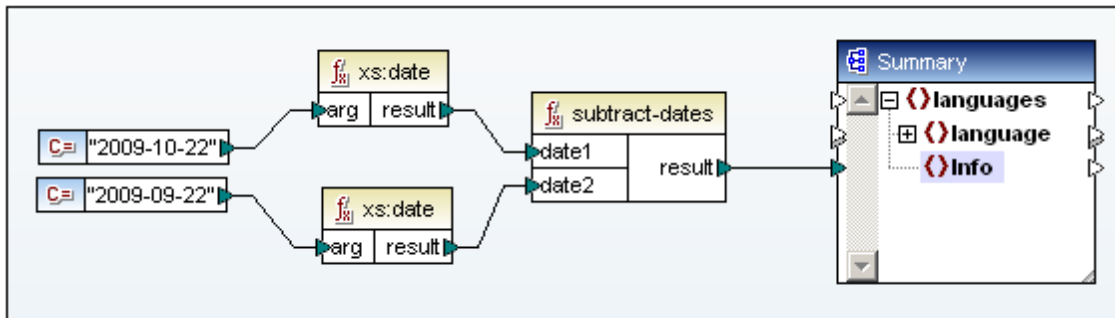




### The 'Subtract' functions

Each of the three subtraction functions enables you to subtract one time value from another and return a duration value. The three subtraction functions are: `subtract-dates`, `subtract-times`, `subtract-dateTimes`.

The screenshot below shows how the `subtract-dates` function is used to subtract two dates (2009-10-22 minus 2009-09-22). The result is the `dayTimeDuration` P30D.



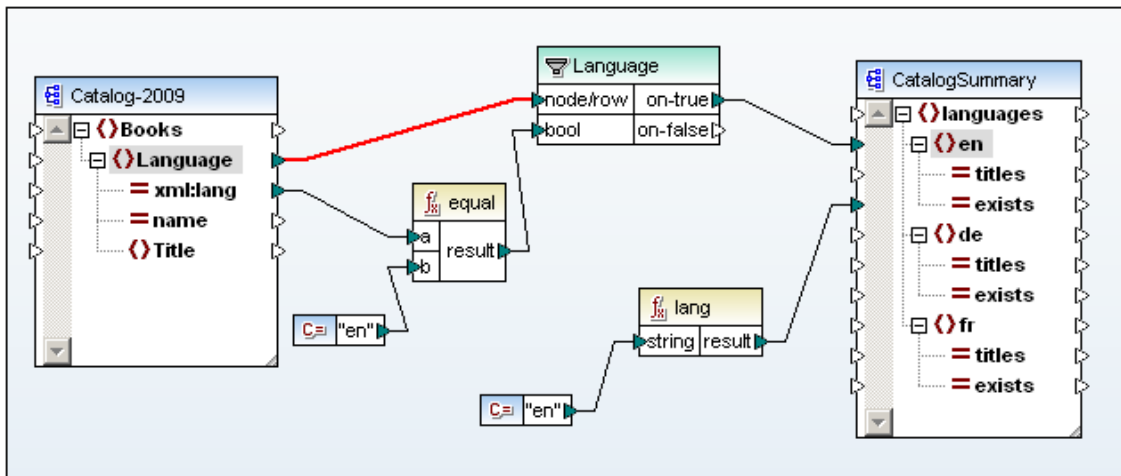
## 7.5.17 xpath2 | node functions

The following XPath 2 node functions are available:

### lang

The `lang` function takes a string argument that identifies a language code (such as `en`). The function returns `true` or `false` depending on whether the context node has an `xml:lang` attribute with a value that matches the argument of the function.





In the screenshot above notice the following:

1. In the source schema, the `Language` element has an `xml:lang` attribute.
2. `Language` nodes are filtered so that only those `Language` nodes having an `xml:lang` value of `en` are processed (the filter test is specified in the `equal` function).
3. The `Language` node is the context node at the point where the `en` element is created in the output document.
4. The output of the `lang` function (`true` or `false`) is sent to the `en/@exists` attribute node of the output. The argument of the function is provided by the string constant `en`. The `lang` function then checks whether the context node at this point (the `Language` element) has an `xml:lang` attribute with a value of `en` (the argument of the function). If yes, then `true` is returned, otherwise `false`.

### local-name, name, namespace-uri

The `local-name`, `name`, and `namespace-uri` functions, return, respectively, the local-name, name, and namespace URI of the input node. For example, for the node `altova:Products`, the local-name is `Products`, the name is `altova:Products`, and the namespace URI is the URI of the namespace to which the `altova:` prefix is bound (say, `http://www.altova.com/mapforce`).

Each of these three functions has two variants:

- With no argument: the function is then applied to the context node (for an example of a context node, see the example given for the `lang` function above).
- An argument that must be a node: the function is applied to the submitted node.

The output of each of these six variants is a string.

### number

Converts an input string into a number. Also converts a boolean input to a number.

The `number` function takes a node as input, atomizes the node (that is, extracts its contents), and converts the value to a decimal and returns the converted value. The only types that can be converted to numbers are booleans, strings, and other numeric types. Non-numeric input values (such as a non-numeric string) result in `NaN` (Not a Number).

There are two variants of the `number` function:



- With no argument: the function is then applied to the context node (for an example of a context node, see the example given for the `lang` function above).
- An argument that must be a node: the function is applied to the submitted node.

## 7.5.18 xpath2 | numeric functions

The following XPath 2 numeric functions are available:

### **abs**

The `abs` function takes a numeric value as input and returns its absolute value as a decimal. For example, if the input argument is `-2` or `+2`, the function returns `2`.

### **round-half-to-even**

The `round-half-to-even` function rounds the supplied number (first argument) to the degree of precision (number of decimal places) supplied in the optional second argument. For example, if the first argument is `2.141567` and the second argument is `3`, then the first argument (the number) is rounded to three decimal places, so the result will be `2.141`. If no precision (second argument) is supplied, the number is rounded to zero decimal places, that is, to an integer.

The 'even' in the name of the function refers to the rounding to an even number when a digit in the supplied number is midway between two values. For example, `round-half-to-even(3.475, 2)` would return `3.48`.

## 7.5.19 xpath2 | string functions

The following XPath 2 string functions are available:

### **compare**

The `compare` function takes two strings as arguments and compares them for equality and alphabetically. If *String-1* is alphabetically less than *String-2* (for example the two string are: `A` and `B`), then the function returns `-1`. If the two strings are equal (for example, `A` and `A`), the function returns `0`. If *String-1* is greater than *String-2* (for example, `B` and `A`), then the function returns `+1`.

A variant of this function allows you to choose what collation is to be used to compare the strings. When no collation is used, the default collation, which is the Unicode codepoint collation, is used. The Altova Engines support the Unicode codepoint collation only.

### **ends-with**

The `ends-with` function tests whether *String-1* ends with *String-2*. If yes, the function returns `true`, otherwise `false`.

A variant of this function allows you to choose what collation is to be used to compare the strings. When no collation is used, the default collation, which is the Unicode codepoint collation, is used. The Altova Engines support the Unicode codepoint collation only.



**escape-uri**

The `escape-uri` function takes a URI as input for the first string argument and applies the URI escaping conventions of RFC 2396 to the string. The second boolean argument (`escape-reserved`) should be set to `true()` if characters with a reserved meaning in URIs are to be escaped (for example "+" or "/").

For example:

```
escape-uri("My A+B.doc", true()) would give My%20A%2B.doc
escape-uri("My A+B.doc", false()) would give My%20A+B.doc
```

**lower-case**

The `lower-case` function takes a string as its argument and converts every upper-case character in the string to its corresponding lower-case character.

**matches**

The `matches` function tests whether a supplied string (the first argument) matches a regular expression (the second argument). The syntax of regular expressions must be that defined for the `pattern` facet of XML Schema. The function returns `true` if the string matches the regular expression, `false` otherwise.

The function takes an optional `flags` argument. Four flags are defined (`i`, `m`, `s`, `x`). Multiple flags can be used: for example, `imx`. If no flag is used, the default values of all four flags are used.

The meaning of the four flags are as follows:

- `i`     Use case-insensitive mode. The default is case-sensitive.
- `m`     Use multiline mode, in which the input string is considered to have multiple lines, each separated by a newline character (`x0a`). The meta characters `^` and `$` indicate the beginning and end of each line. The default is string mode, in which the string starts and ends with the meta characters `^` and `$`.
- `s`     Use dot-all mode. The default is not-dot-all mode, in which the meta character `.` matches all characters except the newline character (`x0a`). In dot-all mode, the dot also matches the newline character.
- `x`     Ignore whitespace. By default whitespace characters are not ignored.

**normalize-unicode**

The `normalize-unicode` function normalizes the input string (the first argument) according to the rules of the normalization form specified (the second argument). The normalization forms NFC, NFD, NFKC, and NFKD are supported.

**replace**

The `replace` function takes the string supplied in the first argument as input, looks for matches as specified in a regular expression (the second argument), and replaces the matches with the string in the third argument.

The rules for matching are as specified for the `matches` attribute above. The function also takes an



optional `flags` argument. The flags are as described in the `matches` function above.

### **starts-with**

The `starts-with` function tests whether *String-1* starts with *String-2*. If yes, the function returns `true`, otherwise `false`.

A variant of this function allows you to choose what collation is to be used to compare the strings. When no collation is used, the default collation, which is the Unicode codepoint collation, is used. The Altova Engines support the Unicode codepoint collation only.

### **substring-after**

The `substring-after` function returns that part of *String-1* (the first argument) that occurs after the test string, *String-2* (the second argument). An optional third argument specifies the collation to use for the string comparison. When no collation is used, the default collation, which is the Unicode codepoint collation, is used. The Altova Engines support the Unicode codepoint collation only.

### **substring-before**

The `substring-before` function returns that part of *String-1* (the first argument) that occurs before the test string, *String-2* (the second argument). An optional third argument specifies the collation to use for the string comparison. When no collation is used, the default collation, which is the Unicode codepoint collation, is used. The Altova Engines support the Unicode codepoint collation only.

### **upper-case**

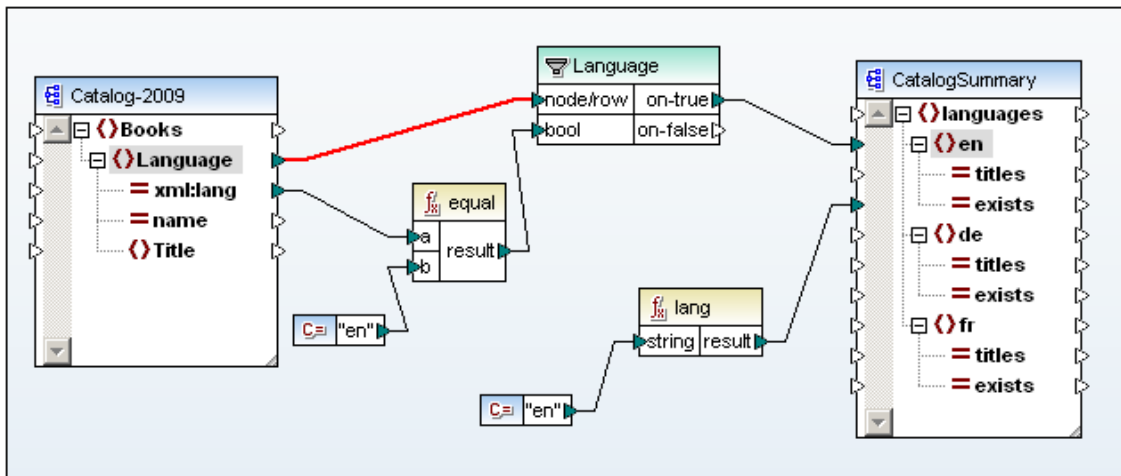
The `upper-case` function takes a string as its argument and converts every lower-case character in the string to its corresponding upper-case character.

## **7.5.20 xslt | xpath functions**

The functions in the XPath Functions library are XPath 1.0 nodeset functions. Each of these functions takes a node or nodeset as its context and returns information about that node or nodeset. These function typically have:

- a context node (in the screenshot below, the context node for the `lang` function is the `Language` element of the source schema).
- an input argument (in the screenshot below, the input argument for the `lang` function is the string constant `en`). The `last` and `position` functions take no argument.





### lang

The `lang` function takes a string argument that identifies a language code (such as `en`). The function returns `true` or `false` depending on whether the context node has an `xml:lang` attribute with a value that matches the argument of the function. In the screenshot above notice the following:

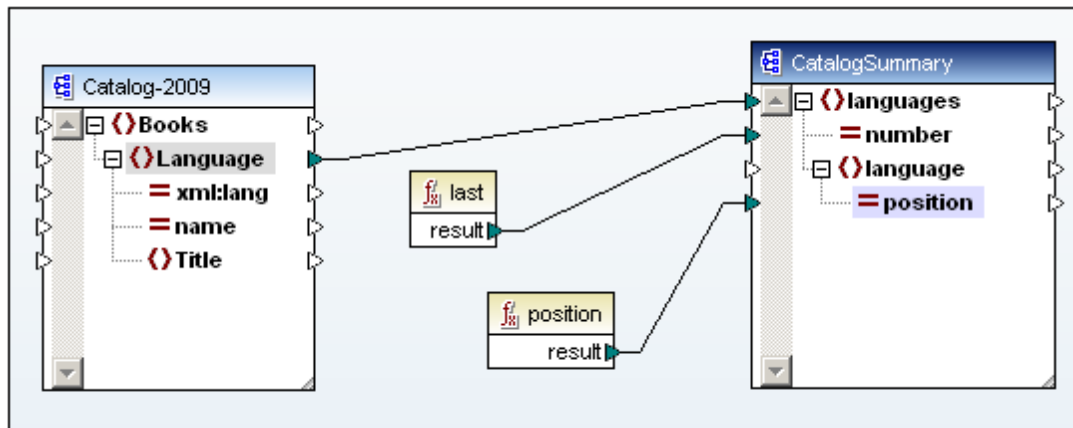
1. In the source schema, the `Language` element has an `xml:lang` attribute.
2. `Language` nodes are filtered so that only those `Language` nodes having an `xml:lang` value of `en` are processed (the filter test is specified in the `equal` function).
3. The `Language` node is the context node at the point where the `en` element is created in the output document.
4. The output of the `lang` function (`true` or `false`) is sent to the `en/@exists` attribute node of the output. The argument of the function is provided by the string constant `en`. The `lang` function then checks whether the context node at this point (the `Language` element) has an `xml:lang` attribute with a value of `en` (the argument of the function). If yes, then `true` is returned, otherwise `false`.

### last, position

The `last` and `position` functions take no argument. The `last` function returns the position of the last node in the context nodeset. The `position` function returns the position of the current node in the nodeset being processed.

The context nodeset at the nodes where the functions are directed is the nodeset to which the functions will apply. In the screenshot below, the nodeset of `Language` elements is the context nodeset for the `last` and `position` functions.





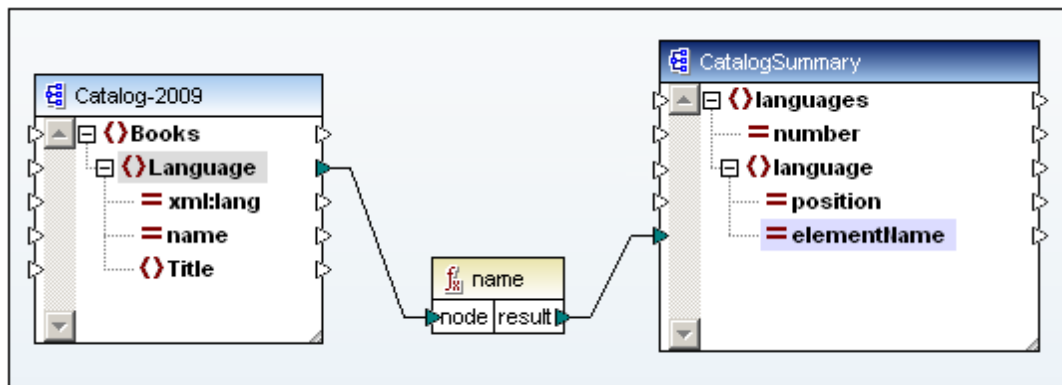
In the example above, the `last` function returns the position of the last node of the context nodeset (the nodeset of `Language` elements) as the value of the `number` attribute. This value is also the size of the nodeset since it indicates the number of nodes in the nodeset.

The `position` function returns the position of the `Language` node being currently processed. For each `Language` element node, its position within the nodeset of `Language` elements is output to the `language/@position` attribute node.

### name, local-name, namespace-uri

These functions are all used the same way and return, respectively, the name, local-name, and namespace URI of the input node. The screenshot below shows how these functions are used. Notice that no context node is specified.

The `name` function returns the name of the `Language` node and outputs it to the `language/@elementname` attribute. If the argument of any of these functions is a nodeset instead of a single node, the name (or local-name or namespace URI) of the first node in the nodeset is returned.



The `name` function returns the QName of the node; the `local-name` function returns the local-name part of the node's QName. For example, if a node's QName is `altova:MyNode`, then `MyNode` is the local name.

The namespace URI is the URI of the namespace to which the node belongs. For example, the `altova:` prefix can be declared to map to a namespace URI in this way:

```
xmlns:altova="http://www.altova.com/namespaces".
```



**Note:** Additional XPath 1.0 functions can be found in the Core function library.

## 7.5.21 xslt | xslt functions

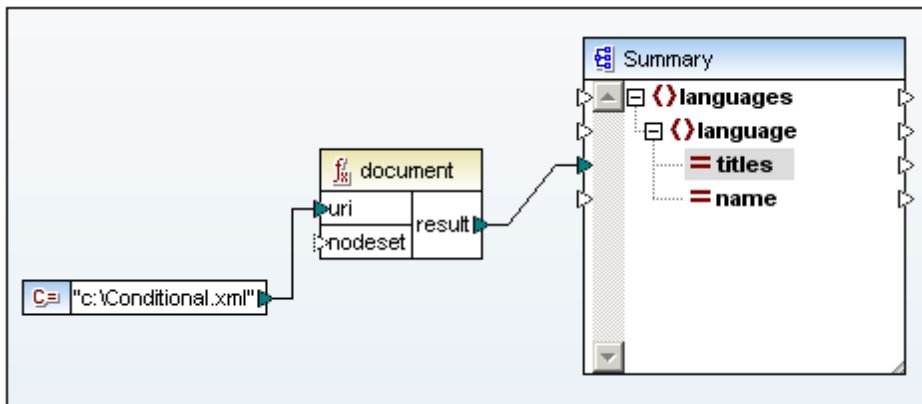
The functions in the XSLT Functions library are XSLT 1.0 functions.

### 7.5.21.1 *current*

The **current** function takes no argument and returns the current node.

### 7.5.21.2 *document*

The **document** function addresses an external XML document (with the `uri` argument; see *screenshot below*). The optional `nodeset` argument specifies a node, the base URI of which is used to resolve the URI supplied as the first argument if this URI is relative. The result is output to a node in the output document.



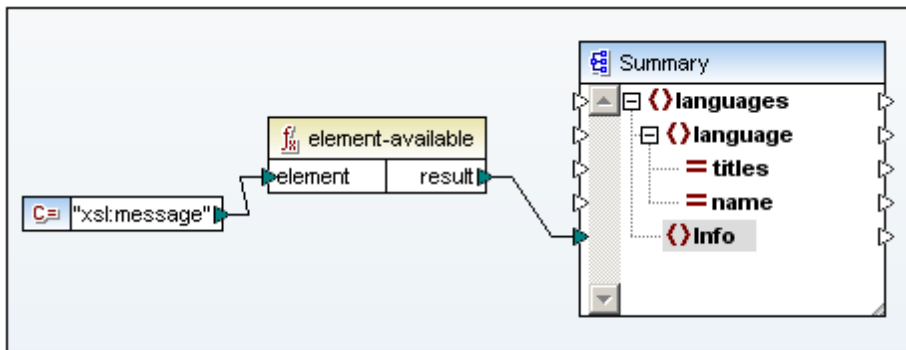
Note that the `uri` argument is a string that must be an absolute file path.

### 7.5.21.3 *element-available*

The **element-available** function tests whether an element, entered as the only string argument of the function, is supported by the XSLT processor.

The argument string is evaluated as a QName. Therefore, XSLT elements must have an `xsl:` prefix and XML Schema elements must have an `xs:` prefix—since these are the prefixes declared for these namespaces in the underlying XSLT that will be generated for the mapping.





The function returns a boolean.

#### 7.5.21.4 *function-available*

The **function-available** function is similar to the **element-available** function and tests whether the function name supplied as the function's argument is supported by the XSLT processor.

The input string is evaluated as a QName. The function returns a boolean.

#### 7.5.21.5 *generate-id*

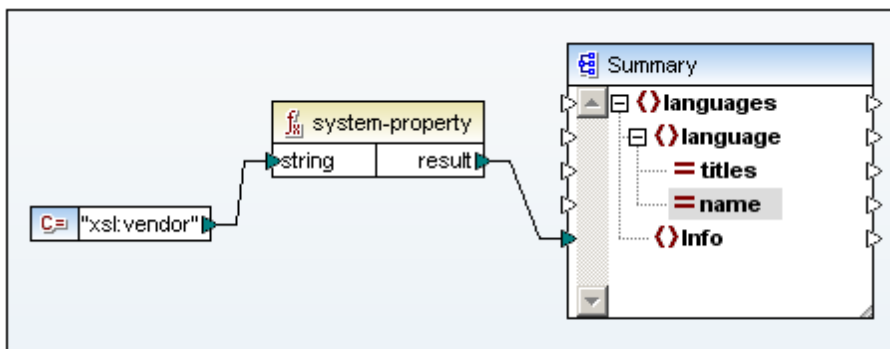
The **generate-id** function generates a unique string that identifies the first node in the nodeset identified by the optional input argument.

If no argument is supplied, the ID is generated on the context node. The result can be directed to any node in the output document.

#### 7.5.21.6 *system-property*

The **system-property** function returns properties of the XSLT processor (the system). Three system properties, all in the XSLT namespace, are mandatory for XSLT processors. These are `xsl:version`, `xsl:vendor`, and `xsl:vendor-url`.

The input string is evaluated as a QName and so must have the `xsl:prefix`, since this is the prefix associated with the XSLT namespace in the underlying XSLT stylesheet.





### 7.5.21.7 *unparsed-entiy-uri*

If you are using a DTD, you can declare an unparsed entity in it. This unparsed entity (for example an image) will have a URI that locates the unparsed entity.

The input string of the function must match the name of the unparsed entity that has been declared in the DTD. The function then returns the URI of the unparsed entity, which can then be directed to a node in the output document, for example, to an `href` node.







# Chapter 8

---

## Automating Mappings and MapForce



## 8 Automating Mappings and MapForce

Mappings designed with MapForce can be executed in a server environment (including Linux and macOS servers), and with server-level performance, by the following Altova transformation engines (licensed separately):

- *RaptorXML Server*. Running a mapping with this engine is suitable if the transformation language of the mapping is XSLT 1.0, XSLT 2.0, or XQuery. See [Automation with RaptorXML Server](#).
- *MapForce Server (or MapForce Server Advanced Edition)*. This engine is suitable for any mapping where the transformation language is BUILT-IN\*. The BUILT-IN language supports the most mapping features in MapForce, while MapForce Server (and, in particular, MapForce Server Advanced Edition) provides best performance for running a mapping.

\* *The BUILT-IN transformation language requires MapForce Professional or Enterprise Edition.*

In addition to this, MapForce provides the ability to automate generation of XSLT code from the command line interface. For more information, see [MapForce Command Line Interface](#).



## 8.1 Automation with RaptorXML Server

RaptorXML Server (hereafter also called RaptorXML for short) is Altova's third-generation, super-fast XML and XBRL processor. It has been built to be optimized for the latest standards and parallel computing environments. Designed to be highly cross-platform capable, the engine takes advantage of today's ubiquitous multi-core computers to deliver lightning fast processing of XML and XBRL data.

RaptorXML is available in two editions which can be downloaded from the Altova download page (<https://www.altova.com/download-trial-server.html>):

- RaptorXML Server is a very fast XML processing engine with support for XML, XML Schema, XSLT, XPath, XQuery, and more. This edition is part of the FlowForce Server installation package.
- RaptorXML+XBRL Server supports all the features of RaptorXML Server with the additional capability of processing and validating the XBRL family of standards.

If you generate code in XSLT 1.0 or 2.0, MapForce creates a batch file called **DoTransform.bat** which is placed in the output folder that you choose upon generation. Executing the batch file calls RaptorXML Server and executes the XSLT transformation on the server.

**Note:** You can also [preview the XSLT](#) code using the built-in engine.



## 8.2 MapForce Command Line Interface

The general syntax of a MapForce command at the command line is:

```
MapForce.exe <filename> [/ {target} [[<outputdir>] [/options]]]
```

### Legend

The following notation is used to indicate command line syntax:

Notation	Description
Text without brackets or braces	Items you must type as shown
<Text inside angle brackets>	Placeholder for which you must supply a value
[Text inside square brackets]	Optional items
{Text inside braces}	Set of required items; choose one
Vertical bar ( )	Separator for mutually exclusive items; choose one
Ellipsis (...)	Items that can be repeated

### <filename>

The mapping design (.mfd) file from which code is to be generated.

### / {target}

Specifies the target language or environment for which code is to be generated. The following code generation targets are supported.

Target	Description
/XSLT	Generates XSLT 1.0 code.
/XSLT2	Generates XSLT 2.0 code.

### <outputdir>

Optional parameter which specifies the output directory. If an output path is not supplied, the current working directory will be used. Note that any relative file paths are relative to the current working directory.



## /options

The /options are not mutually exclusive. One or more of the following options can be set.

Option	Description
/GLOBALRESOURCEFILE <filename>	<p>This option is applicable if the mapping uses Global Resources to resolve input or output file or folder paths, or databases. For more information, see <a href="#">Altova Global Resources</a>.</p> <p>The option /GLOBALRESOURCEFILE specifies the path to a Global Resource .xml file. Note that, if /GLOBALRESOURCEFILE is set, then /GLOBALRESOURCECONFIG must also be set.</p>
/GLOBALRESOURCECONFIG <config>	<p>This option specifies the name of the Global Resource configuration (see also the previous option). Note that, if /GLOBALRESOURCEFILE is set, then /GLOBALRESOURCECONFIG must also be set.</p>
/LOG <logfilename>	<p>Generates a log file at the specified path. &lt;logfilename&gt; can be a full path name, for example, it can include both a directory and a file name. However, if a full path is supplied, the directory must exist for the log file to be generated. If you only specify the file name, then the file will be placed in the &lt;outputdir&gt; directory.</p>

## Remarks

- Relative paths are relative to the working directory, which is the current directory of the application calling MapForce. This applies to the path of the .mfd filename, output directory, log filename, and global resource filename.
- Do not use the end backslash and closing quote at the command line (for example, "C:\My directory\"). These two characters are interpreted by the command line parser as a literal double quotation mark. Use the double backslash \\ if spaces occur in the command line and you need the quotes ("c:\My Directory\\"), or try to avoid using spaces and therefore quotes at all.

## Examples

1) To start MapForce and open the mapping <filename>.mfd, use:

```
MapForce.exe <filename>.mfd
```

2) To generate XSLT 2.0 code and also create a log file with the name <logfilename>, use:

```
MapForce.exe <filename>.mfd /XSLT2 <outputdir> /LOG <logfilename>
```

3) To generate XSLT 2.0 code taking into account the global resource configuration



<grconfigname> from the global resource file <grfilename>, use:

```
Mapforce.exe <filename>.mfd /XSLT2 <outputdir> /GLOBALRESOURCEFILE  
<grfilename> /GLOBALRESOURCECONFIG <grconfigname>
```



# Chapter 9

---

## Customizing MapForce



## 9 Customizing MapForce

This section provides information about working with Altova Global Resources, and working with catalog files.



## 9.1 Changing the MapForce Options

You can change the general and other preferences in MapForce as follows:

- On the **Tools** menu, click **Options**.

The available options are grouped as shown below.

### Libraries

From this page, you can add or delete custom function libraries to MapForce. For more information, see [Importing Custom XSLT 1.0 or 2.0 Functions](#).

### General

The settings available in this page are as follows:

<b>Show logo   Show on start</b>	Shows or hides an image (splash screen) while MapForce starts.
<b>Show gradient background</b>	Enables or disables the gradient background in the Mapping pane.
<b>Limit annotation display to N lines</b>	This option applies to components which support annotations (for example, XML schema, EDI). If the annotation text contains multiple lines, then enabling this option shows only the first N lines on the component, where N is the value you specify. This setting also applies to SELECT statements visible in a component.
<b>Encoding name</b>	Sets the default character encoding for new components. This setting can also be changed individually for each component, see <a href="#">Changing the Component Settings</a> .
<b>Use execution timeout</b>	Sets an execution timeout when previewing the mapping result in the <b>Output</b> pane.
<b>Generate output to temporary files</b>	<p>When this option is set, the output generated when you preview the mapping result will be written to temporary files (this is the default option). If the output file path contains folders that do not exist yet, MapForce will create these folders.</p> <p><b>Warning:</b> If you intend to deploy the mapping to a server for execution, any directories in the path must exist on the server; otherwise, an execution error will occur. See also Preparing Mappings for Server Execution.</p>
<b>Write directly to final output files</b>	<p>When this option is set, the output generated when you preview the mapping result will be written to actual files. If the output file path contains folders that do not exist yet, then a mapping error occurs.</p> <p><b>Warning:</b> This option overwrites any existing output files without requesting further confirmation.</p>



<b>Show logo   Show on start</b>	Shows or hides an image (splash screen) while MapForce starts.
<b>Display text in steps of N million characters</b>	Specifies the maximum size of the text displayed in the <b>Output</b> pane when you preview mappings that generate large XML and text files. If the output text exceeds this value, you will need click a <b>Load more</b> button to load the next chunk. For more information, see <a href="#">Previewing the Output</a> .

### Editing

The settings available in this page are as follows:

<b>Align components on mouse dragging</b>	Specify whether components or functions should be aligned with other components, while you drag them with the mouse, see <a href="#">Aligning Components</a> .
<b>Smart component deletion</b>	When enabled, this option "remembers" connections of deleted components, see <a href="#">Keeping Connections After Deleting Components</a> .

### Messages

From this page, you can re-enable message notifications that were previously disabled using the "Do not show this message again" option.

### Network proxy

See [Network Proxy Settings](#).



## 9.2 Altova Global Resources

Altova Global Resources represent a way to refer to files, folders, or databases so as to make these resources reusable, configurable and available across multiple Altova applications. For example, let's assume that several MapForce mappings routinely read data from the same XML file which is critical for your business workflow. If this file has been renamed on the disk for whatever reason, this would cause "file not found" errors in multiple contexts, and break the workflow. To prevent such issues, it is possible to create a so-called "file alias" (in other words, a Global Resource), and change all mappings to refer to this Global Resource instead of the actual file on disk. This way, if the file name ever changes, you would only need to change the file alias, in one place.

Global Resources can be defined and shared between the following Altova desktop applications: Authentic, MobileTogether Designer, MapForce, DatabaseSpy, and XMLSpy. On the server side, Global Resources can be consumed by the following Altova server applications: MapForce Server, MapForce Server Advanced Edition, RaptorXML Server, RaptorXML+XBRL Server.

Global Resources (be they file, folder, or database references) can be used in MapForce for various scenarios, for example:

- To supply a configurable file path as mapping input, see [Example: Run Mappings with Variable Input Files](#).
- To redirect the mapping output to a configurable path. For more information, see [Example: Generate Mapping Output to Variable Folders](#).

**Note:**

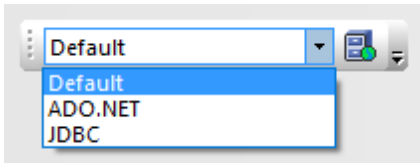
- FlowForce Server does not support Global Resources. MapForce Server can consume Global Resources either at the command line or at API level.
- MapForce Basic Edition does not support consuming database connections defined as Global Resources.

### 9.2.1 Creating Global Resources

A Global Resource alias is a reusable reference which represents a file or folder path, or a database connection. Aliases are defined only once and can be reused as many times as necessary in contexts which support them, including across multiple Altova applications. Taking databases as example, if you frequently work with a specific database in more than one Altova application, then it is a good idea to add the database connection as a Global Resource. This way, you wouldn't need to go through all the Database Connection Wizard steps each time when you need to connect to the same database from another Altova application.

File, folder, and database aliases are configurable in their turn, by means of so-called "configurations". Configurations make it possible to easily switch between files, folders and databases that are consumed or produced by Altova applications, which is particularly useful for testing scenarios. For example, you could create a database alias that consists of three separate connections to the same database, each with a different driver kind: (a) ODBC, the default connection kind, (b) JDBC, and (c) ADO.NET. This way, to connect to the database with a specific driver, you would just select the corresponding configuration from the Global Resources drop-down list before running the mapping.








*Global Resources drop-down list*

Configurations can also help you generate mapping output to variable folders, with a click of a button. For example, you could create a folder alias with two configurations: (a) "Testing", which points to directory **C:\Testing** and (b) "Production", which points to directory **C:\Production**. It is then possible to configure a mapping to generate output to either **C:\Testing** or **C:\Production** folders, just by selecting the required configuration from the Global Resources drop-down list before running the mapping. This example is discussed in more detail in [Example: Generate Output to Variable Folders](#).

### How to create a Global Resource alias

1. On the **Tools** menu, click **Global Resources**. (Alternatively, click the **Global Resource**  toolbar button.)
2. Click **Add** and select the resource type you wish to create (file, folder, database).
3. Enter a descriptive name for this alias in the **Resource alias** text box (for example, "MappingInputFile", "MappingOutputFolder", "DatabaseConnection").
4. Set up the "Default" configuration:
  - a) If it's a file or folder, browse for the file or folder to which this resource should point by default.
  - b) If it's a database connection, click **Choose Database** and follow the Database Connection Wizard to connect to the database. This database connection will be used by default when the mapping runs (unless a different configuration is explicitly selected from the Global Resources drop-down list or supplied as a command line parameter in server execution).
5. Optionally, if the resource should have an additional configuration (for example, a driver kind in case of databases, or an alternative path in case of files or folders), click the **Add configuration**  button, enter a descriptive name (for example "ProductionFolder" or "JDBC\_Alternative"), and set it up as follows:
  - a) If it's a file or folder, browse for the file or folder to which this resource should point as an alternative to the default configuration defined in previous step.
  - b) If it's a database connection, follow the Database Connection Wizard to connect to the database. This database connection will be used as an alternative to the default one.
 In some cases, it might be more convenient to create a configuration as a copy of the default configuration, and then edit it. In this case, click the **Add configuration as a copy of the currently selected configuration**  button.
6. Repeat the previous step for each additional configuration required.


## 9.2.2 The Global Resources XML File

By default, all Global Resources, regardless of the Altova application where they were created, are stored at the following path: **C:\Users\Documents\Altova\GlobalResources.xml**. This makes them transparent, easy to backup, as well as portable to other workstations where Altova



products are installed. It is also possible to rename or duplicate the **GlobalResources.xml** file and thus create multiple Global Resource files. However, only one Global Resource file can be active at a time in an Altova application.

**To set up the active Global Resource file:**

1. On the **Tools** menu, click **Global Resources**. (Alternatively, click the **Global Resource**  toolbar button.)
2. Click **Browse** and select the required Global Resource XML file.

If you are using multiple Global Resource files, make sure that the currently active Global Resource file contains all Global Resources required to run the mapping. For example, if a mapping was configured to read data from a path using a Global Resource, then the currently active Global Resource file must contain that specific Global Resource. Otherwise, error messages like "Errors resolving global resource" will occur in the **Messages** window.

### 9.2.3 Example: Run Mapping with Variable Input Files


Let's assume that, as part of your job duties, you frequently run a mapping that takes as input an XML file. Under normal circumstances, whenever you want to change the input XML of the mapping, you can open the properties of the source XML component and browse for the new input file, see [Changing the Component Settings](#). This is easy to accomplish if it's a one time task. However, what if you need to change the input XML file of the mapping multiple times per day, or even per hour? For example, every morning you need to run the mapping and generate a report by using one XML file as mapping input, and every evening the same report must be generated from another XML file. This is where Global Resources can help you: instead of editing the mapping multiple times per day (or keeping multiple copies of it), you could configure the mapping to read from a file defined as a global resource (a so-called "file alias"). To address the requirement laid out in this example, the file alias could be configured to have two configurations:

1. "Default" - This configuration would supply a "morning" XML file as mapping input
2. "EveningReports" - This configuration would supply an "evening" XML file as mapping input.


Having these configurations in place would make it possible to run the mapping with either input file. Once the file alias is set up as shown below, you will be able to select the desired configuration from a drop-down list, before running the mapping.

#### Step 1: Create the Global Resource

The file alias can be created as follows:

1. On the **Tools** menu, click **Global Resources**. (Alternatively, click the **Global Resource**  toolbar button.)
2. Click **Add | File**.
3. Enter a name in the **Resource alias** text box (in this example, "DailyReports" would be an appropriate name).
4. Click **Browse** and select the following file: **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\mf-ExpReport.xml**.

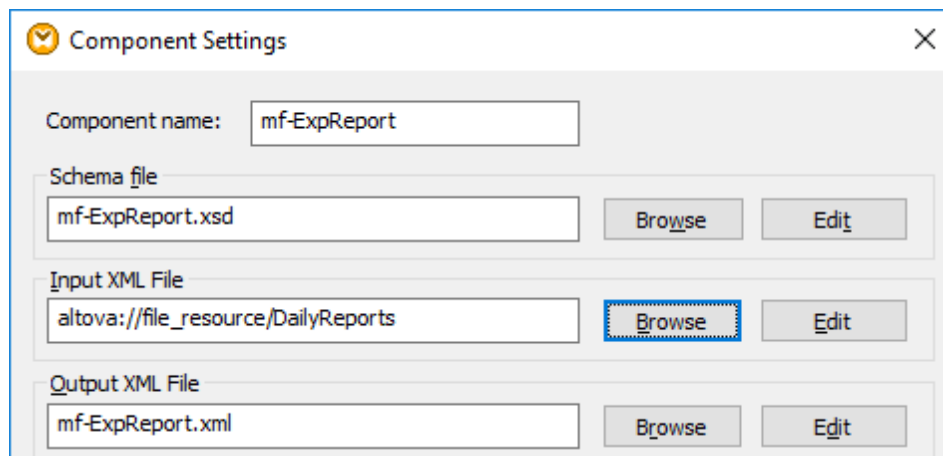


5. Click **Add Configuration**  and name it "EveningReports".
6. Click **Browse** and this time select the following file: <Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\mf-ExpReport2.xml.

## Step 2: Use the Global Resource in the mapping

The required Global Resource has now been created; however, the mapping is not using it yet. To change the mapping so that it reads from the previously defined file alias (Global Resource), do the following:

1. Open the following mapping <Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\Tut-ExpReport.mfd.
2. Right-click the header of the source component on the mapping, and select **Properties** from the context menu.
3. Next to **Input XML file**, click **Browse**.
4. Click **Switch to Global Resources** and select the file alias "DailyReports" defined previously.
5. Click **Open**. The input XML file path has now become **altova://file\_resource/DailyReports**, which indicates that the path uses a Global Resource.



## Step 3: Run the mapping with the desired configuration

You can now easily switch the input XML file before running the mapping, as follows:

- On the **Tools** menu, click **Active Configuration | Default**, to use the file **mf-ExpReport.xml** as input.
- On the **Tools** menu, click **Active Configuration | EveningReports**, to use the file **mf-ExpReport2.xml** as input.

Alternatively, select the required configuration from the **Global Resources** drop-down list.



To preview the mapping result with either configuration, click the **Output** tab and observe differences in the generated output.



## 9.2.4 Example: Generate Output to Variable Folders

This example illustrates how mapping output can be redirected to different folders by means of Global Resources.



Let's suppose that sometimes you need to generate the mapping output to one directory (for example, **C:\Testing**), while in certain cases output must be generated to another directory (for example, **C:\Production**). With Global Resources, this is possible by creating a folder alias with two configurations:

1. "Default" configuration - Generates output to **C:\Testing**
2. "Production" configuration - Generates output to **C:\Production**.

The steps below illustrate how to achieve this goal.

### Step 1: Create the Global Resource

The folder alias can be created as follows:

1. On the **Tools** menu, click **Global Resources**. (Alternatively, click the **Global Resource**  toolbar button.)
2. Click **Add | Folder**.
3. Enter a name in the **Resource alias** text box (in this example, "OutputDirectory" could be an appropriate name).
4. Click **Browse** and select the following folder: **C:\Testing**. (Make sure that this folder already exists on your operating system.)
5. Click **Add Configuration**  and enter a name for the new configuration (in this example, "ProductionDirectory").
6. Click **Browse** and this time select the following folder: **C:\Production**. (Make sure that this folder already exists on your operating system.)

### Step 2: Use the Global Resource in the mapping

The required Global Resource has now been created; however, the mapping is not using it yet. To change the mapping so that it uses from the previously defined folder alias (Global Resource), do the following:

1. Open the following mapping **<Documents>\Altova\MapForce2018\MapForceExamples\Tutorial\Tut-ExpReport.mfd**.
2. Right-click the target component on the mapping, and select **Properties** from the context menu.
3. Next to **Output XML file**, click **Browse**.
4. Click **Switch to Global Resources**, and then click **Save**.
5. When prompted to save the output XML file, enter **output.xml** (or another descriptive file name that you wish to give to the output file). The output XML file path has now become **altova://folder\_resource/OutputDirectory/output.xml**, which indicates that the path is defined as a Global Resource.



### Step 3: Run the mapping with the desired configuration

You can now easily switch to the desired mapping output folder file before running the mapping, as follows:

- On the **Tools** menu, click **Active Configuration | Default**, and then click the **Output** tab to preview the mapping result. The mapping output (either a temporary or a permanent file, as explained below) will be generated in the **C:\Testing** directory.
- On the **Tools** menu, click **Active Configuration | ProductionDirectory**, and then click the **Output** tab. The mapping output (either a temporary or a permanent file, as explained below) will be generated in the **C:\Production** directory.

**Note:** The mapping output is written by default as a temporary file, unless you explicitly configured MapForce to write output to permanent files.

To configure MapForce to generate permanent files instead of temporary, do the following:

1. On the **Tools** menu, click **Options**.
2. In the **General** section, select the option **Write directly to final output files**.



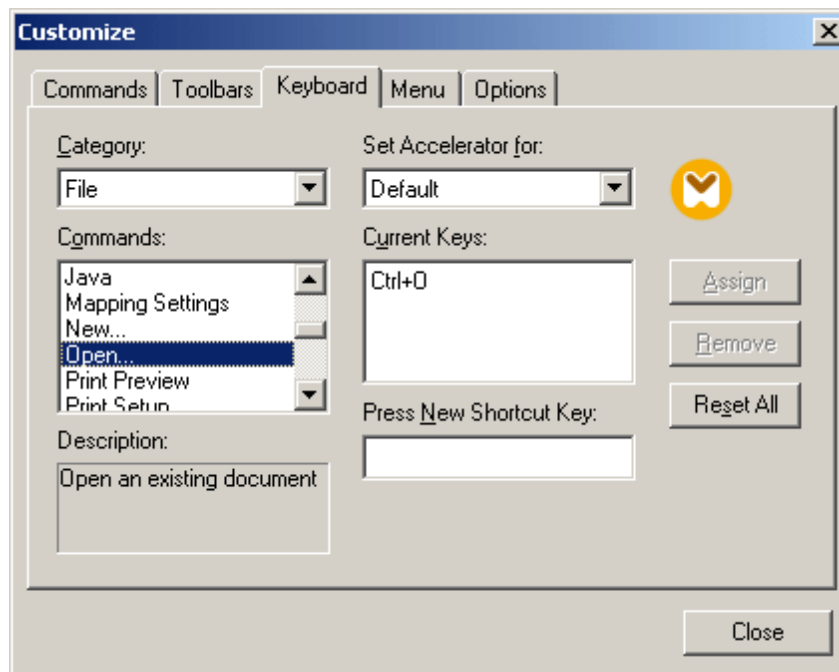
## 9.3 Customizing Keyboard Shortcuts

You can define or change the keyboard shortcuts in MapForce as follows:

1. On the **Tools** menu, click **Customize**.
2. Click the **Keyboard** tab.

**To assign a new Shortcut to a command:**

1. Select the **Tools | Customize** command and click the Keyboard tab.
2. Click the **Category** combo box to select the menu name.
3. Select the **command** you want to assign a new shortcut to, in the Commands list box
4. Click in the **Press New Shortcut Key:** text box, and press the shortcut keys that are to activate the command.



The shortcuts appear immediately in the text box. If the shortcut was assigned previously, then that function is displayed below the text box.

5. Click the **Assign** button to assign the shortcut.  
The shortcut now appears in the Current Keys list box.  
(To **clear** the entry in the Press New Shortcut Key text box, press any of the control keys, **CTRL**, **ALT** or **SHIFT**).

**To de-assign or delete a shortcut:**

1. Click the shortcut you want to delete in the Current Keys list box.
2. Click the **Remove** button.
3. Click the **Close** button to confirm.

**Note:** The **Set accelerator for** does not currently have any function.



The currently assigned keyboard shortcuts are as follows:

F1	Help Menu
F2	Next bookmark (in output window)
F3	Find Next
F10	Activate menu bar
Num +	Expand current item node
Num -	Collapse item node
Num *	Expand all from current item node
CTRL + TAB	Switches between open mappings
CTRL + F6	Cycle through open windows
CTRL + F4	Closes the active mapping document
Alt + F4	Closes MapForce
Alt + F, F, 1	Opens the last file
Alt + F, T, 1	Opens the last project
CTRL + N	File New
CTRL + O	File Open
CTRL + S	File Save
CTRL + P	File Print
CTRL + A	Select All
CTRL + X	Cut
CTRL + C	Copy
CTRL + V	Paste
CTRL + Z	Undo
CTRL + Y	Redo
Del	Delete component (with prompt)
Shift + Del	Delete component (no prompt)
CTRL + F	Find
F3	Find Next
Shift + F3	Find Previous
<b>Arrow keys</b>	
(up / down)	Select next item of component
Esc	Abandon edits/close dialog box
Return	Confirms a selection
<b>Output window hotkeys</b>	
CTRL + F2	Insert Remove/Bookmark
F2	Next Bookmark
SHIFT + F2	Previous Bookmark
CTRL + SHIFT + F2	Remove All Bookmarks
<b>Zooming hotkeys</b>	
CTRL + mouse wheel forward	Zoom In
CTRL + mouse wheel back	Zoom Out
CTRL + 0 (Zero)	Reset Zoom



## 9.4 Catalog Files

MapForce supports a subset of the OASIS XML catalogs mechanism. The catalog mechanism enables MapForce to retrieve commonly used schemas (as well as stylesheets and other files) from local user folders. This increases the overall processing speed, enables users to work offline (that is, not connected to a network), and improves the portability of documents (because URIs would then need to be changed only in the catalog files.)

The catalog mechanism in MapForce works as outlined below.

### RootCatalog.xml

When MapForce starts, it loads a file called `RootCatalog.xml` (structure shown in listing below), which contains a list of catalog files that will be looked up. You can modify this file and enter as many catalog files to look up as you like, each in a `nextCatalog` element. Each of these catalog files is looked up and the URIs in them are resolved according to the mappings specified in them.

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
  xmlns:spy="http://www.altova.com/catalog_ext"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:entity:xmlns:xml:catalog
Catalog.xsd">
  <nextCatalog catalog="%PersonalFolder%/Altova/%AppAndVersionName%/
CustomCatalog.xml"/>
  <nextCatalog catalog="CoreCatalog.xml"/>
  <!-- Include all catalogs under common schemas folder on the first directory
level -->
  <nextCatalog spy:recurseFrom="%AltovaCommonFolder%/Schemas"
catalog="catalog.xml" spy:depth="1"/>
  <!-- Include all catalogs under common XBRL folder on the first directory
level -->
  <nextCatalog spy:recurseFrom="%AltovaCommonFolder%/XBRL"
catalog="catalog.xml" spy:depth="1"/>
</catalog>
```

In the listing above, notice that in the `Schemas` and `XBRL` folders of the folder identified by the variable `%AltovaCommonFolder%` are catalog files named `catalog.xml`. (The value of the `%AltovaCommonFolder%` variable is given in the table below.)

The catalog files in the Altova Common Folder map the pre-defined public and system identifiers of commonly used schemas (such as SVG and WSDL) and XBRL taxonomies to URIs that point to locally saved copies of the respective schemas. These schemas are installed in the Altova Common Folder when MapForce is installed. You should take care not to duplicate mappings in these files, as this could lead to errors.

### CoreCatalog.xml, CustomCatalog.xml, and Catalog.xml

In the `RootCatalog.xml` listing above, notice that `CoreCatalog.xml` and `CustomCatalog.xml` are listed for lookup:

- `CoreCatalog.xml` contains certain Altova-specific mappings for locating schemas in the



Altova Common Folder.

- `CustomCatalog.xml` is a skeleton file in which you can create your own mappings. You can add mappings to `CustomCatalog.xml` for any schema you require but that is not addressed by the catalog files in the Altova Common Folder. Do this using the supported elements of the OASIS catalog mechanism (see *below*).
- There are a number of `Catalog.xml` files in the Altova Common Folder. Each is inside the folder of a specific schema or XBRL taxonomy in the Altova Common Folder, and each maps public and/or system identifiers to URIs that point to locally saved copies of the respective schemas.

### Location of catalog files and schemas

The files `RootCatalog.xml` and `CoreCatalog.xml` are installed in the MapForce application folder. The file `CustomCatalog.xml` is located in your `MyDocuments/Altova/MapForce` folder. The `catalog.xml` files are each in a specific schema folder, these schema folders being inside the folders: `%AltovaCommonFolder%\Schemas` and `%AltovaCommonFolder%\XBRL`.

### Shell environment variables and Altova variables

Shell environment variables can be used in the `nextCatalog` element to specify the path to various system locations (see *RootCatalog.xml* listing above). The following shell environment variables are supported:

<code>%AltovaCommonFolder%</code>	C:\Program Files\Altova\Common2018
<code>%DesktopFolder%</code>	Full path to the Desktop folder for the current user.
<code>%ProgramMenuFolder%</code>	Full path to the Program Menu folder for the current user.
<code>%StartMenuFolder%</code>	Full path to Start Menu folder for the current user.
<code>%StartUpFolder%</code>	Full path to Start Up folder for the current user.
<code>%TemplateFolder%</code>	Full path to the Template folder for the current user.
<code>%AdminToolsFolder%</code>	Full path to the file system directory that stores administrative tools for the current user.
<code>%AppDataFolder%</code>	Full path to the Application Data folder for the current user.
<code>%CommonAppDataFolder%</code>	Full path to the file directory containing application data for all users.
<code>%FavoritesFolder%</code>	Full path of the Favorites folder for the current user.
<code>%PersonalFolder%</code>	Full path to the Personal folder for the current user.
<code>%SendToFolder%</code>	Full path to the SendTo folder for the current user.



%FontsFolder%	Full path to the System Fonts folder.
%ProgramFilesFolder%	Full path to the Program Files folder for the current user.
%CommonFilesFolder%	Full path to the Common Files folder for the current user.
%WindowsFolder%	Full path to the Windows folder for the current user.
%SystemFolder%	Full path to the System folder for the current user.
%CommonAppDataFolder%	Full path to the file directory containing application data for all users.
%LocalAppDataFolder%	Full path to the file system directory that serves as the data repository for local (nonroaming) applications.
%MyPicturesFolder%	Full path to the MyPictures folder.

### How catalogs work

Catalogs are commonly used to redirect a call to a DTD to a local URI. This is achieved by mapping, in the catalog file, public or system identifiers to the required local URI. So when the DOCTYPE declaration in an XML file is read, the public or system identifier locates the required local resource via the catalog file mapping.

For popular schemas, the PUBLIC identifier is usually pre-defined, thus requiring only that the URI in the catalog file point to the correct local copy. When the XML document is parsed, the PUBLIC identifier in it is read. If this identifier is found in a catalog file, the corresponding URL in the catalog file will be looked up and the schema will be read from this location. So, for example, if the following SVG file is opened in MapForce:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="20" height="20" xml:space="preserve">
  <g style="fill:red; stroke:#000000">
    <rect x="0" y="0" width="15" height="15"/>
    <rect x="5" y="5" width="15" height="15"/>
  </g>
</svg>
```

This document is read and the catalog is searched for the PUBLIC identifier. Let's say the catalog file contains the following entry:



```
<catalog>
...
  <public publicId="-//W3C//DTD SVG 1.1//EN" uri="schemas/svg/svg11.dtd"/>
...
</catalog>
```

In this case, there is a match for the `PUBLIC` identifier, so the lookup for the SVG DTD is redirected to the URI `schemas/svg/svg11.dtd` (this path is relative to the catalog file), and this local file will be used as the DTD. If there is no mapping for the `Public` ID in the catalog, then the URL in the XML document will be used (in the example above: `http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd`).

### The catalog subset supported by MapForce

When creating entries in `CustomCatalog.xml` (or any other catalog file that is to be read by MapForce), use only the following elements of the OASIS catalog specification. Each of the elements below is listed with an explanation of their attribute values. For a more detailed explanation, see the [XML Catalogs specification](#). Note that each element can take the `xml:base` attribute, which is used to specify the base URI of that element.

- `<public publicId="PublicID of Resource" uri="URL of local file"/>`
- `<system systemId="SystemID of Resource" uri="URL of local file"/>`
- `<uri name="filename" uri="URL of file identified by filename"/>`
- `<rewriteURI uriStartString="StartString of URI to rewrite" rewritePrefix="String to replace StartString"/>`
- `<rewriteSystem systemIdStartString="StartString of SystemID" rewritePrefix="Replacement string to locate resource locally"/>`

In cases where there is no public identifier, as with most stylesheets, the system identifier can be directly mapped to a URL via the `system` element. Also, a URI can be mapped to another URI using the `uri` element. The `rewriteURI` and `rewritesSystem` elements enable the rewriting of the starting part of a URI or system identifier, respectively. This allows the start of a filepath to be replaced and consequently enables the targeting of another directory. For more information on these elements, see the [XML Catalogs specification](#).

### File extensions and intelligent editing according to a schema

Via catalog files you can also specify that documents with a particular file extension should have MapForce's intelligent editing features applied in conformance with the rules in a schema you specify. For example, if you create a custom file extension `.myhtml` for (HTML) files that are to be valid according to the HTML DTD, then you can enable intelligent editing for files with these extensions by adding the following element of text to `CustomCatalog.xml` as a child of the `<catalog>` element.

```
<spy:fileExtHelper ext="myhtml" uri="schemas/xhtml/xhtml11-transitional.dtd"/>
```

This would enable intelligent editing (auto-completion, entry helpers, etc) of `.myhtml` files in MapForce according to the XHTML 1.0 Transitional DTD. Refer to the `catalog.xml` file in the `%AltovaCommonFolder%\Schemas\/xhtml` folder, which contains similar entries.

### XML Schema and catalogs

XML Schema information is built into MapForce and the validity of XML Schema documents is



checked against this internal information. In an XML Schema document, therefore, no references should be made to any schema for XML Schema.

The `catalog.xml` file in the `%AltovaCommonFolder%\Schemas\schemas` folder contains references to DTDs that implement older XML Schema specifications. You should not validate your XML Schema documents against either of these schemas. The referenced files are included solely to provide MapForce with entry helper info for editing purposes should you wish to create documents according to these older recommendations.

**More information**

For more information on catalogs, see the [XML Catalogs specification](#).



## 9.5 Network Proxy Settings

The **Network Proxy** section enables you to configure custom proxy settings. These settings affect how the application connects to the Internet (for XML validation purposes, for example). By default, the application uses the system's proxy settings, so you should not need to change the proxy settings in most cases. If necessary, however, you can set an alternative network proxy using the options below.

**Note:** The network proxy settings are shared between all Altova MissionKit applications. Consequently, if you change the settings in one application, they will automatically affect all other applications.

**Network Proxy**

☒ Use system proxy settings

☐ Automatic proxy configuration

☒ Auto-detect settings

Script URL

☐ Manual proxy configuration

HTTP Proxy  Port

☐ Use this proxy server for all protocols

SSL Proxy  Port

No Proxy for

☐ Do not use the proxy server for local addresses

Current proxy settings

Test URL

Found IE auto-proxy configuration.  
Methods WPAD (using test URL http://www.example.com)  
PAC resolved DIRECT (NO PROXY).  
Using no Proxy.

### Use system proxy settings

Uses the Internet Explorer (IE) settings configurable via the system proxy settings. It also queries the settings configured with `netsh.exe winhttp`.

### Automatic proxy configuration

The following options are provided:

- *Auto-detect settings:* Looks up a WPAD script (`http://wpad.LOCALDOMAIN/wpad.dat`) via DHCP or DNS, and uses this script for proxy setup.



- *Script URL*: Specify an HTTP URL to a proxy-auto-configuration (.pac) script that is to be used for proxy setup.
- *Reload*: Resets and reloads the current auto-proxy-configuration. This action requires Windows 8 or newer, and may need up to 30s to take effect.

### Manual proxy configuration

Manually specify the fully qualified host name and port for the proxies of the respective protocols. A supported scheme may be included in the host name (for example: `http://hostname`). It is not required that the scheme is the same as the respective protocol if the proxy supports the scheme.

The following options are provided:

- *Use this proxy for all protocols*: Uses the host name and port of the HTTP Proxy for all protocols.
- *No Proxy for*: A semi-colon (;) separated list of fully qualified host names, domain names, or IP addresses for hosts that should be used without a proxy. IP addresses may not be truncated and IPv6 addresses have to be enclosed by square brackets (for example: `[2606:2800:220:1:248:1893:25c8:1946]`). Domain names must start with a leading dot (for example: `.example.com`).
- *Do not use the proxy server for local addresses*: If checked, adds `<local>` to the *No Proxy for* list. If this option is selected, then the following will not use the proxy: (i) `127.0.0.1`, (ii) `:::1`, (iii) all host names not containing a dot character (.).

### Current proxy settings

Provides a verbose log of the proxy detection. It can be refreshed with the **Refresh** button to the right of the *Test URL* field (for example, when changing the test URL, or when the proxy settings have been changed).

- *Test URL*: A test URL can be used to see which proxy is used for that specific URL. No I/O is done with this URL. This field must not be empty if proxy-auto-configuration is used (either through *Use system proxy settings* or *Automatic proxy configuration*).







# Chapter 10

---

## Menu Reference



## 10 Menu Reference

This reference section contains a description of the MapForce menu commands.



## 10.1 File

**New**

Creates a new mapping document.

**Open**

Opens previously saved mapping design (.mfd) files. Note that it is not possible to open mapping files which contain features not available in your MapForce edition.

**Save**

Saves the currently active mapping using the currently active file name.

**Save As**

Saves the currently active mapping with a different name, or allows you to supply a new name if this is the first time you save it.

**Save All**

Saves all currently open mapping files.

**Reload**

Reloads the currently active mapping file. You are asked if you want to lose your last changes.

**Close**

Closes the currently active mapping file. You are asked if you want to save the file before it closes.

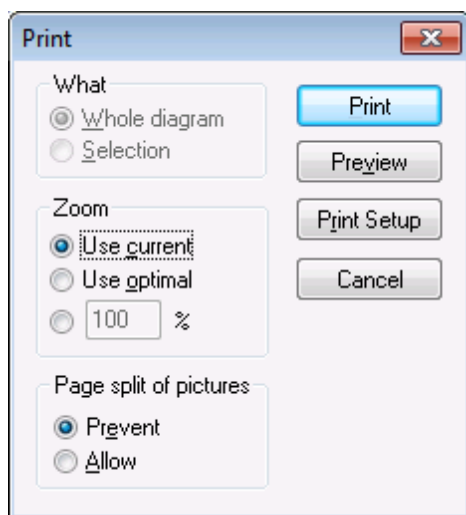
**Close All**

Closes all currently open mapping files. You are asked if you want to save any of the unsaved mapping files.

**Print**

Opens the Print dialog box, from where you can print out your mapping as hard copy.





*Print dialog box*

**Use current** retains the currently defined zoom factor of the mapping. **Use optimal** scales the mapping to fit the page size. You can also specify the zoom factor numerically. Component scrollbars are not printed. You can also specify if you want to allow the graphics to be split over several pages or not.

### Print Preview

Opens the same Print dialog box with the same settings as described above.

### Print Setup

Opens the Print Setup dialog box in which you can define the printer you want to use and the paper settings.

### Validate Mapping

Validates that all mappings (connectors) are valid and displays any warnings or errors (see [Validating mappings](#) ).

### Mapping settings

Opens the Mapping Settings dialog box where you can define the document-specific settings (see [Changing the mapping settings](#) ).

### Generate code in selected language

Generates code in the currently selected language of your mapping. The currently selected language is visible as a highlighted programming language icon in the toolbar: XSLT, XSLT 2.

### Generate code in | XSLT (XSLT2)

This command generates the XSLT file(s) needed for the transformation from the source file(s). Selecting this option opens the Browse for Folder dialog box where you select the location of the XSLT file. The name of the generated XSLT file(s) is defined in the Mapping Settings dialog box (see [Changing the mapping settings](#) ).

### Recent files

Displays a list of the most recently opened files.



**Exit**

Exits the application. You are asked if you want to save any unsaved files.



## 10.2 Edit

Most of the commands in this menu become active when you view the result of a mapping in the **Output** tab, or preview XSLT code in the XSLT tab.

### Undo

MapForce has an unlimited number of "Undo" steps that you can use to retrace you mapping steps.

### Redo

The redo command allows you to redo previously undone commands. You can step backward and forward through the undo history using both these commands.

### Find

Allows you to search for specific text in either the XSLT, XSLT2 or Output tab.

### Find Next **F3**

Searches for the next occurrence of the same search string.

### Find Previous **Shift F3**

Searches for the previous occurrence of the same search string.

### Cut/Copy/Paste/Delete

The standard windows Edit commands, allow you to cut, copy etc., any components or functions visible in the mapping window.

### Select all

Selects all components in the Mapping tab, or the text/code in the XSLT, XSLT2, or Output tab.



## 10.3 Insert

### XML Schema / File

Adds to the mapping an XML schema or instance file. If you select an XML file which references a schema, no additional information is required for the mapping. If you select an XML file without a schema reference, you are prompted to generate a matching XML schema automatically (see [Generating an XML Schema](#)). If you select an XML schema file, you are prompted to include optionally an XML instance file which supplies the data for preview.

### Insert Input

When the mapping window displays a mapping, this command adds an input component to the mapping (see [Supplying Parameters to the Mapping](#)). When the mapping window displays a user-defined function, this command adds an input component to the user-defined function (see [Defining Complex Input Components](#)).

### Insert Output

When the mapping window displays a mapping, this command adds an output component to the mapping (see [Returning String Values from a Mapping](#)). When the mapping window displays a user-defined function, this command adds an output component to the user-defined function (see [Defining Complex Output Components](#)).

### Constant

Inserts a constant which supplies fixed data to an input [connector](#). The data is entered into a dialog box when creating the component. You can select the following types of data: String, Number and All other.

### Variable

Inserts an Intermediate Variable which is equivalent to a regular (non-inline) user-defined function. Variables are structural components, without instance files, and are used to simplify the mapping process (see [Intermediate variables](#)).

### Sort: Nodes/Rows

Inserts a component which allows you to sort nodes (see [Sort Nodes/Rows](#)).

### Filter: Nodes/Rows

Inserts a component that uses two input and output parameters: **node/row** and **bool**, and **on-true**, **on-false**. If the Boolean is true, then the value of the node/row parameter is forwarded to the on-true parameter. If the Boolean is false, then the complement value is passed on to the on-false parameter. For more information, see [Filters and Conditions](#).

### Value-Map

Inserts a component that transforms an input value to an output value using a lookup table. This is useful when you need to map a set of values to another set of values (for example, month numbers to month names). For more information, see [Using Value-Maps](#).

### IF-Else Condition

Inserts a component of type "If-Else Condition" (see [Filters and Conditions](#)).







## 10.4 Component

### **Change Root Element**

Allows you to change the root element of the XML instance document.

### **Edit Schema Definition in XMLSpy**

Selecting this option, having previously clicked an XML-Schema/document, opens the XML Schema file in the Schema view of XMLSpy where you can edit it.

### **Add Duplicate Input Before**

Inserts a copy/clone of the selected item before the currently selected item. Duplicate items do not have output icons, you cannot use them as data sources. For an example, see [Map Multiple Sources to One Target](#) section in the tutorial. Right clicking a duplicate item also allows you to reposition it using the menu items Move Up/Move Down, depending on where the item is.

### **Add Duplicate Input After**

Inserts a copy/clone of the selected item after the currently selected item. Duplicate items do not have output icons, you cannot use them as data sources. For an example, see the [Map Multiple Sources to One Target](#) section in the tutorial. Right clicking a duplicate item also allows you to reposition it using the menu items Move Up/Move Down, depending on where the item is.

### **Remove Duplicate**

Removes a previously defined duplicate item. For an example, see the [Map Multiple Sources to One Target](#) section in the tutorial.

### **Align Tree Left**

Aligns all the items along the left hand window border.

### **Align Tree Right**

Aligns all the items along the right hand window border. This display is useful when creating mappings to the target schema.

### **Properties**

Opens a dialog box which displays the settings of the currently selected component. See [Changing the Component Settings](#) .



## 10.5 Connection


### **Auto Connect Matching Children**

Activates or deactivates the "Auto Connect Matching Children" option, as well as the icon in the icon bar.

### **Settings for Connect Matching Children**

Opens the Connect Matching Children dialog box in which you define the connection settings (see [Connecting matching children](#)).

### **Connect Matching Children**

This command allows you to create multiple connectors for items of the **same name**, in both the source and target schemas. The settings you define in this dialog box are retained, and are applied when connecting two items, if the "**Auto connect child items**" icon  in the title bar is active. Clicking the icon switches between an active and inactive state. For further information, see [Connecting matching children](#).

### **Target Driven (Standard)**

Changes the connector type to Standard mapping. For further information, see [Target Driven \(Standard\) mapping](#).

### **Copy-all (Copy Child Items)**

Creates connectors for all matching child items, where each of the child connectors are displayed as a subtree of the parent connector (see [Copy-all connections](#) ).

### **Source Driven (Mixed Content)**

Changes the connector type to *Source Driven (Mixed Content)*. For further information, see [Source Driven \(Mixed Content\) mapping](#).

### **Properties**

Opens a dialog box in which you can define the specific (mixed content) settings of the current connector. Unavailable options are greyed out. These settings also apply to **complexType** items which do not have any text nodes. For further information, see [Connection settings](#).



## 10.6 Function

### Create User-Defined Function

Creates a new user-defined function (see [User-defined functions](#)).

### Create User-Defined Function from Selection

Creates a new user-defined function based on the currently selected elements in the mapping window.

### Function Settings

Opens the settings dialog box of the currently active user-defined function allowing you to change its settings.

### Remove Function

Deletes the currently active user-defined function if you are working in a context which allows this.

### Insert Input

When the mapping window displays a mapping, this command adds an input component to the mapping (see [Simple Input](#) ). When the mapping window displays a user-defined function, this command adds an input component to the user-defined function (see [Defining Complex Input Components](#) ).

### Insert Output

When the mapping window displays a mapping, this command adds an output component to the mapping (see [Simple Output](#) ). When the mapping window displays a user-defined function, this command adds an output component to the user-defined function (see [Defining Complex Output Components](#) ).



## 10.7 Output

### **XSLT 1.0, XSLT 2.0, XQuery, Java, C#, C++, Built-in Execution Engine**

Sets the transformation language in which the mapping should be executed (see [Selecting a Transformation Language](#)).

### **Validate Output File**

Validates the output XML file against the referenced schema (see [Validating the Mapping Output](#)).

### **Save Output File**

Saves the data visible in the Output pane to a file.

### **Save All Output Files**

Saves all the generated output files of dynamic mappings. See [Processing Multiple Input or Output Files Dynamically](#) for more information.

### **Regenerate Output**

Regenerates the data visible in the Output pane.

### **Insert/Remove Bookmark**

Inserts a bookmark at the cursor position in the Output pane.

### **Next Bookmark**

Navigates to the next bookmark in the Output pane.

### **Previous Bookmark**

Navigates to the previous bookmark in the Output pane.

### **Remove All Bookmarks**

Removes all currently defined bookmarks in the Output pane.

### **Pretty-Print XML Text**

Reformats your XML document in the Output pane to give a structured display of the document. Each child node is offset from its parent by a single tab character. This is where the Tab size settings (i.e. inserting as tabs or spaces) defined in the Tabs group, take effect.

### **Text View Settings**

Displays the Text View settings dialog box. This dialog box allows you to customize the text view settings in the **Output** pane and **XSLT** pane, and also shows the currently defined hotkeys that apply in the window. For more information, see [Text View Features](#).



## 10.8 View

### Show Annotations

Displays XML schema annotations in the component window.

If the Show Types icon is also active, then both sets of info are show in grid form.

= F1060	
type	string
ann.	Revision identifier

### Show Types

Displays the schema datatypes for each element or attribute.

If the Show Annotations icon is also active, then both sets of info are show in grid form.

### Show library in Function Header

Displays the library name in parenthesis in the function title.

### Show Tips

Displays a tooltip containing explanatory text when the mouse pointer is placed over a function.

### Show Selected Component Connectors

Switches between showing all mapping connectors, or those connectors relating to the currently selected components.

### Show Connectors from Source to Target

Switches between showing:

- connectors that are **directly** connected to the currently selected component, or
- connectors linked to the currently selected component, originating from source and terminating at the target components.

### Zoom

Opens the Zoom dialog box. You can enter the zoom factor numerically, or drag the slider to change the zoom factor interactively.

### Back

Steps back through the currently open mappings of the mapping tab.

### Forward

Steps forward through the currently open mappings of the mapping tab.

### Status Bar

Switches on/off the Status Bar visible below the Messages window.

### Library Window

Switches on/off the Library window.

### Messages

Switches on/off the Validation output window. When generating code the Messages output



window is automatically activated to show the validation result.

**Overview**

Switches on/off the Overview window. Drag the rectangle to navigate your Mapping view.



## 10.9 Tools

### Global Resources

Opens the Manage Global Resources dialog box, where you can add, edit or delete settings applicable across multiple Altova applications (see [Altova Global Resources](#)).

### Active Configuration

Allows you to select the currently active global resource configuration from a list of configurations previously defined in the Global Resources.

### Create Reversed Mapping

Creates a "reversed" mapping from the currently active mapping in MapForce, which is to be the basis of a new mapping. Note that the result is not intended to be a complete mapping, only the direct connections between components are retained in the reversed mapping. It is very likely that the resulting mapping will not be valid or suitable for preview in the Output pane, without manual editing.

When you reverse a mapping, the source component becomes the target component, and target component becomes the source. If an input or output XML instance file have been assigned to a component, then they will be swapped.

The following data is retained:

- Direct connections between components
- Direct connections between components in a chained mapping
- The type of connection: Standard, Mixed content, Copy-All
- Pass-through component settings
- Database components

The following data is not retained:

- Connections via functions, filters, etc, along with the functions, filters, etc.
- User-defined functions
- Web service components

### Restore Toolbars and Windows

Resets the toolbars, entry helper windows, docked windows etc. to their defaults. MapForce needs to be restarted for the changes to take effect.

### Customize...

Opens a dialog box that lets you to customize the MapForce graphical user interface. This includes showing or hiding toolbars, as well as editing the context menus and keyboard shortcuts (see [Customizing Keyboard Shortcuts](#)).

### Options

Opens a dialog box where you can change the default MapForce settings (see [Changing the MapForce Options](#)).



## 10.10 Window

### **Cascade**

This command rearranges all open document windows so that they are all cascaded (i.e. staggered) on top of each other.

### **Tile Horizontal**

This command rearranges all open document windows as **horizontal tiles**, making them all visible at the same time.

### **Tile Vertical**

This command rearranges all open document windows as **vertical tiles**, making them all visible at the same time.

### **1** **2**

This list shows all currently open windows, and lets you quickly switch between them. You can also use the Ctrl-TAB or CTRL F6 keyboard shortcuts to cycle through the open windows.



## 10.11 Help Menu

### ▼ Table of Contents

#### ☐ Description

Opens the onscreen help manual of MapForce with the Table of Contents displayed in the left-hand-side pane of the Help window. The Table of Contents provides an overview of the entire Help document. Clicking an entry in the Table of Contents takes you to that topic.

### ▼ Index

#### ☐ Description

Opens the onscreen help manual of MapForce with the Keyword Index displayed in the left-hand-side pane of the Help window. The index lists keywords and lets you navigate to a topic by double-clicking the keyword. If a keyword is linked to more than one topic, a list of these topics is displayed.

### ▼ Search

#### ☐ Description

Opens the onscreen help manual of MapForce with the Search dialog displayed in the left-hand-side pane of the Help window. To search for a term, enter the term in the input field, and press **Return**. The Help system performs a full-text search on the entire Help documentation and returns a list of hits. Double-click any item to display that item.

---

### ▼ Software Activation

#### ☐ Description

After you download your Altova product software, you can license—or activate—it using either a free evaluation key or a purchased permanent license key.

- **Free evaluation key.** When you first start the software after downloading and installing it, the Software Activation dialog will pop up. In it is a button to request a free evaluation key-code. Enter your name, company, and e-mail address in the dialog that appears, and click Request Now! The evaluation key is sent to the e-mail address you entered and should reach you in a few minutes. Now enter the key in the key-code field of the Software Activation dialog box and click **OK** to start working with your Altova product. The software will be unlocked for a period of 30 days.
- **Permanent license key.** The Software Activation dialog contains a button to purchase a permanent license key. Clicking this button takes you to Altova's online shop, where you can purchase a permanent license key for your product. There are two types of permanent license: single-user and multi-user. Both will be sent to you by e-mail. A *single-user license* contains your license-data and



includes your name, company, e-mail, and key-code. A *multi-user license* contains your license-data and includes your company name and key-code. Note that your license agreement does not allow you to install more than the licensed number of copies of your Altova software on the computers in your organization (per-seat license). Please make sure that you enter the data required in the registration dialog exactly as given in your **license e-mail**.

**Note:** When you enter your license information in the Software Activation dialog, ensure that you enter the data exactly as given in your license e-mail. For multi-user licenses, each user should enter his or her own name in the Name field.

#### Your license email and the different ways to license (activate) your Altova product

The license email that you receive from Altova will contain:

- Your license details (name, company, email, key-code)
- As an attachment, a license file with a `.altova_licenses` file extension

To activate your Altova product, you can do one of the following:

- Enter the email-supplied license details in the Altova product's Software Activation dialog, and click **OK**.
- Save the license file (`.altova_licenses`) to a suitable location, double-click the license file, enter any requested details in the dialog that appears, and finish by clicking **Apply Keys**.
- Save the license file (`.altova_licenses`) to any suitable location, and upload it from this location to the license pool of your Altova LicenseServer. You can then either: (i) acquire the license from your Altova product via the product's Software Activation dialog, or (ii) assign the license to the product from Altova LicenseServer. *For more information about licensing via LicenseServer, read the rest of this topic.*

The Software Activation dialog (*screenshot below*) can be accessed at any time by clicking the **Help | Software Activation** command.

You can activate the software by either:

- Entering the license key information (click **Enter a New Key Code**), or
- Acquiring a license via an Altova LicenseServer on your network (click **Use Altova LicenseServer**, located at the bottom of the Software Activation dialog). The Altova LicenseServer must have a license for your Altova product in its license pool. If a license is available in the LicenseServer pool, this is indicated in the Software Activation dialog (*screenshot below*), and you can click **Save** to acquire the license.



Altova XMLSpy Enterprise Edition 2017 Software Activation

Thank you for choosing Altova XMLSpy Enterprise Edition 2017 and welcome to the software activation process. You can view your assigned license or select an Altova LicenseServer which provides a license for you. (NOTE: To use this software you must be licensed via Altova LicenseServer or a valid license key code from Altova.)

If you do not want to use Altova LicenseServer click here to enter a key code manually Enter Key Code

To activate your software please enter or select the name of the Altova LicenseServer on your network.

Altova LicenseServer: DOC.co ↻

☒ A license is already assigned to you on LicenseServer at DOC.co.

Name	AQA (Concurrent 50 Users)
Company	Altova GmbH
User count	50
License type	concurrent
Expires in	-
SMP	248 days left

Return License Check out License Copy Support Code Save Close

Connected to Altova LicenseServer at DOC.altova.com

After a machine-specific (aka installed) license has been acquired from a LicenseServer, it cannot be returned to the LicenseServer for a period of seven days. After that time, you can return the machine license to LicenseServer (click **Return License**) so that this license can be acquired from LicenseServer by another client. (A LicenseServer administrator, however, can unassign an acquired license at any time via the administrator's Web UI of LicenseServer.) Note that the returning of licenses applies only to machine-specific licenses, not to concurrent licenses.

#### Check out license

You can check out a license from the license pool for a period of up to 30 days so that the license is stored on the product machine. This enables you to work offline, which is useful, for example, if you wish to work in an environment where there is no access to your Altova LicenseServer (such as when your Altova product is installed on a laptop and you are traveling). While the license is checked out, LicenseServer displays the license as being in use, and the license cannot be used by any other machine. The license automatically reverts to the checked-in state when the check-out period ends. Alternatively, a checked-out license can be checked in at any time via the **Check in** button of the Software Activation dialog.

To check out a license, do the following: (i) In the Software Activation dialog, click **Check out License** (see screenshot above); (ii) In the License Check-out dialog that appears, select the check-out period you want and click **Check out**. The license will be checked out. The Software Activation dialog will display the check-out information, including the time when the check-out period ends. The **Check out License** button in the dialog changes to a **Check In** button. You can check the license in again at any time by clicking **Check In**. Because the license automatically reverts to the checked-in status, make sure that the check-out period you select adequately covers the period during which you will be working offline.



**Note:** For license check-outs to be possible, it must be enabled on the LicenseServer. If this functionality has not been enabled, you will get an error message to this effect. In this event, contact your LicenseServer administrator.

Copy Support Code

Click **Copy Support Code** to copy license details to the clipboard. This is the data that you will need to provide when requesting support via the [online support form](#).

Altova LicenseServer provides IT administrators with a real-time overview of all Altova licenses on a network, together with the details of each license, as well as client assignments and client usage of licenses. The advantage of using LicenseServer therefore lies in administrative features it offers for large-volume Altova license management. Altova LicenseServer is available free of cost from the [Altova website](#). For more information about Altova LicenseServer and licensing via Altova LicenseServer, see the [Altova LicenseServer documentation](#).

▼ Order Form

▢ Description

When you are ready to order a licensed version of the software product, you can use either the **Order license key** button in the Software Activation dialog (see *previous section*) or the **Help | Order Form** command to proceed to the secure Altova Online Shop.

▼ Registration

▢ Description

Opens the Altova Product Registration page in a tab of your browser. Registering your Altova software will help ensure that you are always kept up to date with the latest product information.

▼ Check for Updates

▢ Description

Checks with the Altova server whether a newer version than yours is currently available and displays a message accordingly.

▼ Support Center

▢ Description

A link to the Altova Support Center on the Internet. The Support Center provides FAQs,



discussion forums where problems are discussed, and access to Altova's technical support staff.

▼ FAQ on the Web

▢ Description

A link to Altova's FAQ database on the Internet. The FAQ database is constantly updated as Altova support staff encounter new issues raised by customers.

▼ Download Components and Free Tools

▢ Description

A link to Altova's Component Download Center on the Internet. From here you can download a variety of companion software to use with Altova products. Such software ranges from XSLT and XSL-FO processors to Application Server Platforms. The software available at the Component Download Center is typically free of charge.

▼ MapForce on the Internet

▢ Description

A link to the [Altova website](#) on the Internet. You can learn more about MapForce and related technologies and products at the [Altova website](#).

▼ MapForce Training

▢ Description

A link to the Online Training page at the [Altova website](#). Here you can select from online courses conducted by Altova's expert trainers.

▼ About MapForce

▢ Description

Displays the splash window and version number of your product. If you are using the 64-bit version of MapForce, this is indicated with the suffix (x64) after the application name. There is no suffix for the 32-bit version.







# Chapter 11

---

## Appendices



# 11 Appendices

These appendices contain technical information about MapForce and important licensing information. Each appendix contains sub-sections as given below:

## Technical Data

- OS and memory requirements
- Altova XML Parser
- Altova XSLT and XQuery Engines
- Unicode support
- Internet usage
- License metering

## License Information

- Electronic software distribution
- Copyrights
- End User License Agreement



## 11.1 Engine information

This section contains information about implementation-specific features of the Altova XML Validator, Altova XSLT 1.0 Engine, Altova XSLT 2.0 Engine, and Altova XQuery Engine.

### 11.1.1 XSLT and XQuery Engine Information

The XSLT and XQuery engines of MapForce follow the W3C specifications closely and are therefore stricter than previous Altova engines—such as those in previous versions of XMLSpy. As a result, minor errors that were ignored by previous engines are now flagged as errors by MapForce.

For example:

- It is a type error (`err:XPTY0018`) if the result of a path operator contains both nodes and non-nodes.
- It is a type error (`err:XPTY0019`) if `E1` in a path expression `E1/E2` does not evaluate to a sequence of nodes.

If you encounter this kind of error, modify either the XSLT/XQuery document or the instance document as appropriate.

This section describes implementation-specific features of the engines, organized by specification:

- [XSLT 1.0](#)
- [XSLT 2.0](#)
- [XQuery 1.0](#)

#### 11.1.1.1 XSLT 1.0

The XSLT 1.0 Engine of MapForce conforms to the World Wide Web Consortium's (W3C's) [XSLT 1.0 Recommendation of 16 November 1999](#) and [XPath 1.0 Recommendation of 16 November 1999](#). Note the following information about the implementation.

##### Notes about the implementation

When the `method` attribute of `xsl:output` is set to HTML, or if HTML output is selected by default, then special characters in the XML or XSLT file are inserted in the HTML document as HTML character references in the output. For instance, the character U+00A0 (the hexadecimal character reference for a non-breaking space) is inserted in the HTML code either as a character reference (`&#160;` or `&#xA0;`) or as an entity reference, `&nbsp;`.



### 11.1.1.2 XSLT 2.0

This section:

- [Engine conformance](#)
- [Backward compatibility](#)
- [Namespaces](#)
- [Schema awareness](#)
- [Implementation-specific behavior](#)

#### Conformance

The XSLT 2.0 engine of MapForce conforms to the World Wide Web Consortium's (W3C's) [XSLT 2.0 Recommendation of 23 January 2007](#) and [XPath 2.0 Recommendation of 14 December 2010](#).

#### Backwards Compatibility

The XSLT 2.0 engine is backwards compatible. The only time the backwards compatibility of the XSLT 2.0 engine comes into effect is when using the XSLT 2.0 engine to process an XSLT 1.0 stylesheet. Note that there could be differences in the outputs produced by the XSLT 1.0 Engine and the backwards-compatible XSLT 2.0 engine.

#### Namespaces

Your XSLT 2.0 stylesheet should declare the following namespaces in order for you to be able to use the type constructors and functions available in XSLT 2.0. The prefixes given below are conventionally used; you could use alternative prefixes if you wish.

Namespace Name	Prefix	Namespace URI
XML Schema types	xs:	<a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>
XPath 2.0 functions	fn:	<a href="http://www.w3.org/2005/xpath-functions">http://www.w3.org/2005/xpath-functions</a>

Typically, these namespaces will be declared on the `xsl:stylesheet` or `xsl:transform` element, as shown in the following listing:

true

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  ...
>
```

The following points should be noted:

- The XSLT 2.0 engine uses the XPath 2.0 and XQuery 1.0 Functions namespace (listed in the table above) as its **default functions namespace**. So you can use XPath 2.0 and XSLT 2.0 functions in your stylesheet without any prefix. If you declare the XPath 2.0



Functions namespace in your stylesheet with a prefix, then you can additionally use the prefix assigned in the declaration.

- When using type constructors and types from the XML Schema namespace, the prefix used in the namespace declaration must be used when calling the type constructor (for example, `xs:date`).
- Some XPath 2.0 functions have the same name as XML Schema datatypes. For example, for the XPath functions `fn:string` and `fn:boolean` there exist XML Schema datatypes with the same local names: `xs:string` and `xs:boolean`. So if you were to use the XPath expression `string('Hello')`, the expression evaluates as `fn:string('Hello')`—not as `xs:string('Hello')`.

### Schema-awareness

The XSLT 2.0 engine is schema-aware. So you can use user-defined schema types and the `xsl:validate` instruction.

### Implementation-specific behavior

Given below is a description of how the XSLT 2.0 engine handles implementation-specific aspects of the behavior of certain XSLT 2.0 functions.

#### `xsl:result-document`

Additionally supported encodings are (the Altova-specific): `x-base16tobinary` and `x-base64tobinary`.

#### `function-available`

The function tests for the availability of in-scope functions (XSLT, XPath, and extension functions).

#### `unparsed-text`

The `href` attribute accepts (i) relative paths for files in the base-uri folder, and (ii) absolute paths with or without the `file://` protocol. Additionally supported encodings are (the Altova-specific): `x-binarytobase16` and `x-binarytobase64`.

#### `unparsed-text-available`

The `href` attribute accepts (i) relative paths for files in the base-uri folder, and (ii) absolute paths with or without the `file://` protocol. Additionally supported encodings are (the Altova-specific): `x-binarytobase16` and `x-binarytobase64`.

**Note:** The following encoding values, which were implemented in earlier versions of RaptorXML's predecessor product, AltovaXML, are now deprecated: `base16tobinary`, `base64tobinary`, `binarytobase16` and `binarytobase64`.

## 11.1.1.3 XQuery 1.0

This section:

- [Engine conformance](#)
- [Schema awareness](#)
- [Encoding](#)



- [Namespaces](#)
- [XML source and validation](#)
- [Static and dynamic type checking](#)
- [Library modules](#)
- [External functions](#)
- [Collations](#)
- [Precision of numeric data](#)
- [XQuery instructions support](#)

### Conformance

The XQuery 1.0 Engine of MapForce conforms to the World Wide Web Consortium's (W3C's) [XQuery 1.0 Recommendation of 14 December 2010](#). The XQuery standard gives implementations discretion about how to implement many features. Given below is a list explaining how the XQuery 1.0 Engine implements these features.

### Schema awareness

The XQuery 1.0 Engine is **schema-aware**.

### Encoding

The UTF-8 and UTF-16 character encodings are supported.

### Namespaces

The following namespace URIs and their associated bindings are pre-defined.

Namespace Name	Prefix	Namespace URI
XML Schema types	xs:	<a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>
Schema instance	xsi:	<a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a>
Built-in functions	fn:	<a href="http://www.w3.org/2005/xpath-functions">http://www.w3.org/2005/xpath-functions</a>
Local functions	local:	<a href="http://www.w3.org/2005/xquery-local-functions">http://www.w3.org/2005/xquery-local-functions</a>

The following points should be noted:

- The XQuery 1.0 Engine recognizes the prefixes listed above as being bound to the corresponding namespaces.
- Since the built-in functions namespace listed above is the default functions namespace in XQuery, the `fn:` prefix does not need to be used when built-in functions are invoked (for example, `string("Hello")` will call the `fn:string` function). However, the prefix `fn:` can be used to call a built-in function without having to declare the namespace in the query prolog (for example: `fn:string("Hello")`).
- You can change the default functions namespace by declaring the `default function namespace` expression in the query prolog.
- When using types from the XML Schema namespace, the prefix `xs:` may be used



without having to explicitly declare the namespaces and bind these prefixes to them in the query prolog. (Example: `xs:date` and `xs:yearMonthDuration`.) If you wish to use some other prefix for the XML Schema namespace, this must be explicitly declared in the query prolog. (Example: `declare namespace alt = "http://www.w3.org/2001/XMLSchema"; alt:date("2004-10-04");`)

- Note that the `untypedAtomic`, `dayTimeDuration`, and `yearMonthDuration` datatypes have been moved, with the CRs of 23 January 2007, from the XPath Datatypes namespace to the XML Schema namespace, so: `xs:yearMonthDuration`.

If namespaces for functions, type constructors, node tests, etc are wrongly assigned, an error is reported. Note, however, that some functions have the same name as schema datatypes, e.g. `fn:string` and `fn:boolean`. (Both `xs:string` and `xs:boolean` are defined.) The namespace prefix determines whether the function or type constructor is used.

### XML source document and validation

XML documents used in executing an XQuery document with the XQuery 1.0 Engine must be well-formed. However, they do not need to be valid according to an XML Schema. If the file is not valid, the invalid file is loaded without schema information. If the XML file is associated with an external schema and is valid according to it, then post-schema validation information is generated for the XML data and will be used for query evaluation.

### Static and dynamic type checking

The static analysis phase checks aspects of the query such as syntax, whether external references (e.g. for modules) exist, whether invoked functions and variables are defined, and so on. If an error is detected in the static analysis phase, it is reported and the execution is stopped.

Dynamic type checking is carried out at run-time, when the query is actually executed. If a type is incompatible with the requirement of an operation, an error is reported. For example, the expression `xs:string("1") + 1` returns an error because the addition operation cannot be carried out on an operand of type `xs:string`.

### Library Modules

Library modules store functions and variables so they can be reused. The XQuery 1.0 Engine supports modules that are stored in **a single external XQuery file**. Such a module file must contain a `module` declaration in its prolog, which associates a target namespace. Here is an example module:

```
module namespace libns="urn:module-library";
declare variable $libns:company := "Altova";
declare function libns:webaddress() { "http://www.altova.com" };
```

All functions and variables declared in the module belong to the namespace associated with the module. The module is used by importing it into an XQuery file with the `import module` statement in the query prolog. The `import module` statement only imports functions and variables declared directly in the library module file. As follows:



```
import module namespace modlib = "urn:module-library" at "modulefilename.xq";

if      ($modlib:company = "Altova")
then    modlib:webaddress()
else    error("No match found.")
```

### External functions

External functions are not supported, i.e. in those expressions using the `external` keyword, as in:

```
declare function hoo($param as xs:integer) as xs:string external;
```

### Collations

The default collation is the Unicode-codepoint collation, which compares strings on the basis of their Unicode codepoint. Other supported collations are the [ICU collations](#) listed [here](#). To use a specific collation, supply its URI as given in the [list of supported collations](#). Any string comparisons, including for the `fn:max` and `fn:min` functions, will be made according to the specified collation. If the collation option is not specified, the default Unicode-codepoint collation is used.

### Precision of numeric types

- The `xs:integer` datatype is arbitrary-precision, i.e. it can represent any number of digits.
- The `xs:decimal` datatype has a limit of 20 digits after the decimal point.
- The `xs:float` and `xs:double` datatypes have limited-precision of 15 digits.

### XQuery Instructions Support

The `Pragma` instruction is not supported. If encountered, it is ignored and the fallback expression is evaluated.

## 11.1.2 XSLT and XPath/XQuery Functions

This section lists Altova extension functions and other extension functions that can be used in XPath and/or XQuery expressions. Altova extension functions can be used with Altova's XSLT and XQuery engines, and provide functionality additional to that available in the function libraries defined in the W3C standards.

### General points

The following general points should be noted:

- Functions from the core function libraries defined in the W3C specifications can be called



without a prefix. That's because the XSLT and XQuery engines read non-prefixed functions as belonging to a default functions namespace which is that specified in the XPath/XQuery functions specifications <http://www.w3.org/2005/xpath-functions>. If this namespace is explicitly declared in an XSLT or XQuery document, the prefix used in the namespace declaration can also optionally be used on function names.

- In general, if a function expects a sequence of one item as an argument, and a sequence of more than one item is submitted, then an error is returned.
- All string comparisons are done using the Unicode codepoint collation.
- Results that are QNames are serialized in the form [prefix:]localname.

### Precision of xs:decimal

The precision refers to the number of digits in the number, and a minimum of 18 digits is required by the specification. For division operations that produce a result of type `xs:decimal`, the precision is 19 digits after the decimal point with no rounding.

### Implicit timezone

When two `date`, `time`, or `dateTime` values need to be compared, the timezone of the values being compared need to be known. When the timezone is not explicitly given in such a value, the implicit timezone is used. The implicit timezone is taken from the system clock, and its value can be checked with the `implicit-timezone()` function.

### Collations

The default collation is the Unicode codepoint collation, which compares strings on the basis of their Unicode codepoint. The engine uses the Unicode Collation Algorithm. Other supported collations are the [ICU collations](#) listed below; to use one of these, supply its URI as given in the table below. Any string comparisons, including for the `max` and `min` functions, will be made according to the specified collation. If the collation option is not specified, the default Unicode-codepoint collation is used.

Language	URIs
da: Danish	da_DK
de: German	de_AT, de_BE, de_CH, de_DE, de_LI, de_LU
en: English	en_AS, en_AU, en_BB, en_BE, en_BM, en_BW, en_BZ, en_CA, en_GB, en_GU, en_HK, en_IE, en_IN, en_JM, en_MH, en_MP, en_MT, en_MU, en_NA, en_NZ, en_PH, en_PK, en_SG, en_TT, en_UM, en_US, en_VI, en_ZA, en_ZW
es: Spanish	es_419, es_AR, es_BO, es_CL, es_CO, es_CR, es_DO, es_EC, es_ES, es_GQ, es_GT, es_HN, es_MX, es_NI, es_PA, es_PE, es_PR, es_PY, es_SV, es_US, es_UY, es_VE
fr: French	fr_BE, fr_BF, fr_BI, fr_BJ, fr_BL, fr_CA, fr_CD, fr_CF, fr_CG, fr_CH, fr_CI, fr_CM, fr_DJ, fr_FR, fr_GA, fr_GN, fr_GP, fr_GQ, fr_KM, fr_LU, fr_MC, fr_MF, fr_MG, fr_ML, fr_MQ, fr_NE, fr_RE, fr_RW, fr_SN, fr_TD, fr_TG
it: Italian	it_CH, it_IT
ja: Japanese	ja_JP



nb: Norwegian Bokmal	nb_NO
nl: Dutch	nl_AW, nl_BE, nl_NL
nn: Nynorsk	nn_NO
pt: Portuguese	pt_AO, pt_BR, pt_GW, pt_MZ, pt_PT, pt_ST
ru: Russian	ru_MD, ru_RU, ru_UA
sv: Swedish	sv_FI, sv_SE

### Namespace axis

The namespace axis is deprecated in XPath 2.0. Use of the namespace axis is, however, supported. To access namespace information with XPath 2.0 mechanisms, use the `in-scope-prefixes()`, `namespace-uri()` and `namespace-uri-for-prefix()` functions.

## 11.1.2.1 Altova Extension Functions

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, <http://www.altova.com/xslt-extensions>, and are indicated in this section with the prefix **altova:**, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

Functions defined in the W3C's XPath/XQuery Functions specifications can be used in: (i) XPath expressions in an XSLT context, and (ii) in XQuery expressions in an XQuery document. In this documentation we indicate the functions that can be used in the former context (XPath in XSLT) with an **XP** symbol and call them XPath functions; those functions that can be used in the latter (XQuery) context are indicated with an **XQ** symbol; they work as XQuery functions. The W3C's XSLT specifications—not XPath/XQuery Functions specifications—also define functions that can be used in XPath expressions in XSLT documents. These functions are marked with an **XSLT** symbol and are called XSLT functions. The XPath/XQuery and XSLT versions in which a function can be used are indicated in the description of the function (*see symbols below*). Functions from the XPath/XQuery and XSLT function libraries are listed without a prefix. Extension functions from other libraries, such as Altova extension functions, are listed with a prefix.

<i>XPath functions (used in XPath expressions in XSLT):</i>	<b>XP1</b> <b>XP2</b> <b>XP3.1</b>
<i>XSLT functions (used in XPath expressions in XSLT):</i>	<b>XSLT1</b> <b>XSLT2</b> <b>XSLT3</b>
<i>XQuery functions (used in XQuery expressions in XQuery):</i>	<b>XQ1</b> <b>XQ3.1</b>

### XSLT functions



XSLT functions can only be used in XPath expressions in an XSLT context (similarly to XSLT 2.0's `current-group()` or `key()` functions). These functions are not intended for, and will not work in, a non-XSLT context (for instance, in an XQuery context). Note that XSLT functions for XBRL can be used only with editions of Altova products that have XBRL support.

**XPath/XQuery functions**

XPath/XQuery functions can be used both in XPath expressions in XSLT contexts as well as in XQuery expressions:

- [Date/Time](#)
- [Geolocation](#)
- [Image-related](#)
- [Numeric](#)
- [Sequence](#)
- [String](#)
- [Miscellaneous](#)

*XSLT Functions*

**XSLT extension functions** can be used in XPath expressions in an XSLT context. They will not work in a non-XSLT context (for instance, in an XQuery context).

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, <http://www.altova.com/xslt-extensions>, and are indicated in this section with the prefix **altova:**, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

XPath functions (used in XPath expressions in XSLT):	<b>XP1</b> <b>XP2</b> <b>XP3.1</b>
XSLT functions (used in XPath expressions in XSLT):	<b>XSLT1</b> <b>XSLT2</b> <b>XSLT3</b>
XQuery functions (used in XQuery expressions in XQuery):	<b>XQ1</b> <b>XQ3.1</b>

**Standard functions**

▼ `distinct-nodes [altova:]`

`altova:distinct-nodes(node()*)` as `node()`\* **XSLT1** **XSLT2** **XSLT3**

Takes a set of one or more nodes as its input and returns the same set minus nodes with duplicate values. The comparison is done using the XPath/XQuery function `fn:deep-equal`.

▢ Examples

- `altova:distinct-nodes(country)` returns all child `country` nodes less those



having duplicate values.

▼ evaluate [altova:]

**altova:evaluate**(XPathExpression as xs:string[, ValueOf\$p1, ... ValueOf\$pN])  
XSLT1 XSLT2 XSLT3

Takes an XPath expression, passed as a string, as its mandatory argument. It returns the output of the evaluated expression. For example: **altova:evaluate**('//Name[1]') returns the contents of the first `Name` element in the document. Note that the expression `//Name[1]` is passed as a string by enclosing it in single quotes.

The **altova:evaluate** function can optionally take additional arguments. These arguments are the values of in-scope variables that have the names `p1`, `p2`, `p3`... `pN`. Note the following points about usage: (i) The variables must be defined with names of the form `pX`, where `X` is an integer; (ii) the **altova:evaluate** function's arguments (see *signature above*), from the second argument onwards, provide the values of the variables, with the sequence of the arguments corresponding to the numerically ordered sequence of variables: `p1` to `pN`: The second argument will be the value of the variable `p1`, the third argument that of the variable `p2`, and so on; (iii) The variable values must be of type `item*`.

▣ Example

```
<xsl:variable name="xpath" select="'$p3, $p2, $p1'" />
<xsl:value-of select="altova:evaluate($xpath, 10, 20, 'hi')" />
outputs "hi 20 10"
```

In the listing above, notice the following:

- The second argument of the **altova:evaluate** expression is the value assigned to the variable `$p1`, the third argument that assigned to the variable `$p2`, and so on.
- Notice that the fourth argument of the function is a string value, indicated by its being enclosed in quotes.
- The **select** attribute of the `xs:variable` element supplies the XPath expression. Since this expression must be of type `xs:string`, it is enclosed in single quotes.

▣ Examples to further illustrate the use of variables

- ```
<xsl:variable name="xpath" select="'$p1'" />
<xsl:value-of select="altova:evaluate($xpath, //Name[1])" />
Outputs value of the first Name element.
```
- ```
<xsl:variable name="xpath" select="'$p1'" />
<xsl:value-of select="altova:evaluate($xpath, '//Name[1]')" />
Outputs "//Name[1]"
```

The **altova:evaluate()** extension function is useful in situations where an XPath expression in the XSLT stylesheet contains one or more parts that must be evaluated dynamically. For example, consider a situation in which a user enters his request for the sorting criterion and this criterion is stored in the attribute `UserReq/@sortkey`. In the



stylesheet, you could then have the expression: `<xsl:sort select="altova:evaluate(..//UserReq/@sortkey)" order="ascending"/>`. The `altova:evaluate()` function reads the `sortkey` attribute of the `UserReq` child element of the parent of the context node. Say the value of the `sortkey` attribute is `Price`, then `Price` is returned by the `altova:evaluate()` function and becomes the value of the `select` attribute: `<xsl:sort select="Price" order="ascending"/>`. If this `sort` instruction occurs within the context of an element called `Order`, then the `Order` elements will be sorted according to the values of their `Price` children. Alternatively, if the value of `@sortkey` were, say, `Date`, then the `Order` elements would be sorted according to the values of their `Date` children. So the sort criterion for `Order` is selected from the `sortkey` attribute at runtime. This could not have been achieved with an expression like: `<xsl:sort select="..//UserReq/@sortkey" order="ascending"/>`. In the case shown above, the sort criterion would be the `sortkey` attribute itself, not `Price` or `Date` (or any other current content of `sortkey`).

**Note:** The static context includes namespaces, types, and functions—but not variables—from the calling environment. The base URI and default namespace are inherited.

#### More examples

- Static variables: `<xsl:value-of select="$i3, $i2, $i1" />`  
*Outputs the values of three variables.*
- Dynamic XPath expression with dynamic variables:  
`<xsl:variable name="xpath" select="'$p3, $p2, $p1'" />`  
`<xsl:value-of select="altova:evaluate($xpath, 10, 20, 30)" />`  
*Outputs "30 20 10"*
- Dynamic XPath expression with no dynamic variable:  
`<xsl:variable name="xpath" select="'$p3, $p2, $p1'" />`  
`<xsl:value-of select="altova:evaluate($xpath)" />`  
*Outputs error: No variable defined for \$p3.*

#### ▼ `encode-for-rtf [altova:]`

```
altova:encode-for-rtf(input as xs:string, preserveallwhitespace as
xs:boolean, preservenewlines as xs:boolean) as xs:string XSLT2 XSLT3
```

Converts the input string into code for RTF. Whitespace and new lines will be preserved according to the boolean value specified for their respective arguments.

[\[ Top \]](#)

## XBRL functions

Altova XBRL functions can be used only with editions of Altova products that have XBRL support.

#### ▼ `xbml-footnotes [altova:]`

```
altova:xbml-footnotes(node()) as node()* XSLT2 XSLT3
```

Takes a node as its input argument and returns the set of XBRL footnote nodes referenced by the input node.



## ▼ xbrl-labels [altova:]

`altova:xbrl-labels(xs:QName, xs:string) as node()*` **XSLT2 XSLT3**

Takes two input arguments: a node name and the taxonomy file location containing the node.  
The function returns the XBRL label nodes associated with the input node.

[\[ Top \]](#)

### *XPath/XQuery Functions: Date and Time*

Altova's date/time extension functions can be used in XPath and XQuery expressions and provide additional functionality for the processing of data held as XML Schema's various date and time datatypes. The functions in this section can be used with Altova's **XPath 3.0** and **XQuery 3.0** engines. They are available in XPath/XQuery contexts.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, <http://www.altova.com/xslt-extensions>, and are indicated in this section with the prefix **altova:**, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

XPath functions (used in XPath expressions in XSLT):	<b>XP1</b> <b>XP2</b> <b>XP3.1</b>
XSLT functions (used in XPath expressions in XSLT):	<b>XSLT1</b> <b>XSLT2</b> <b>XSLT3</b>
XQuery functions (used in XQuery expressions in XQuery):	<b>XQ1</b> <b>XQ3.1</b>

## ▼ Grouped by functionality

- [Add a duration to xs:dateTime and return xs:dateTime](#)
- [Add a duration to xs:date and return xs:date](#)
- [Add a duration to xs:time and return xs:time](#)
- [Format and retrieve durations](#)
- [Remove timezone from functions that generate current date/time](#)
- [Return weekday as integer from date](#)
- [Return week number as integer from date](#)
- [Build date, time, or duration type from lexical components of each type](#)
- [Construct date, dateTime, or time type from string input](#)
- [Age-related functions](#)

## ▼ Grouped alphabetically

[altova:add-days-to-date](#)



[altova:add-days-to-dateTime](#)  
[altova:add-hours-to-dateTime](#)  
[altova:add-hours-to-time](#)  
[altova:add-minutes-to-dateTime](#)  
[altova:add-minutes-to-time](#)  
[altova:add-months-to-date](#)  
[altova:add-months-to-dateTime](#)  
[altova:add-seconds-to-dateTime](#)  
[altova:add-seconds-to-time](#)  
[altova:add-years-to-date](#)  
[altova:add-years-to-dateTime](#)  
[altova:age](#)  
[altova:age-details](#)  
[altova:build-date](#)  
[altova:build-duration](#)  
[altova:build-time](#)  
[altova:current-dateTime-no-TZ](#)  
[altova:current-date-no-TZ](#)  
[altova:current-time-no-TZ](#)  
[altova:format-duration](#)  
[altova:parse-date](#)  
[altova:parse-dateTime](#)  
[altova:parse-duration](#)  
[altova:parse-time](#)  
[altova:weekday-from-date](#)  
[altova:weekday-from-dateTime](#)  
[altova:weeknumber-from-date](#)  
[altova:weeknumber-from-dateTime](#)

[\[ Top \]](#)

### Add a duration to `xs:dateTime` **XP3.1 XQ3.1**

These functions add a duration to `xs:dateTime` and return `xs:dateTime`. The `xs:dateTime` type has a format of CCYY-MM-DDThh:mm:ss.sss. This is a concatenation of the `xs:date` and `xs:time` formats separated by the letter T. A timezone suffix+01:00 (for example) is optional.

#### ▼ add-years-to-dateTime [altova:]

```
altova:add-years-to-dateTime(DateTime as xs:dateTime, Years as xs:integer) as
xs:dateTime XP3.1 XQ3.1
```

Adds a duration in years to an `xs:dateTime` (*see examples below*). The second argument is the number of years to be added to the `xs:dateTime` supplied as the first argument. The result is of type `xs:dateTime`.

#### ▣ Examples

- `altova:add-years-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), 10)`  
returns `2024-01-15T14:00:00`
- `altova:add-years-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), -4)`  
returns `2010-01-15T14:00:00`

#### ▼ add-months-to-dateTime [altova:]



```
altova:add-months-to-dateTime(DateTime as xs:dateTime, Months as xs:integer) as xs:dateTime XP3.1 XQ3.1
```

Adds a duration in months to an `xs:dateTime` (see examples below). The second argument is the number of months to be added to the `xs:dateTime` supplied as the first argument. The result is of type `xs:dateTime`.

▢ Examples

- `altova:add-months-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), 10)`  
returns `2014-11-15T14:00:00`
- `altova:add-months-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), -2)`  
returns `2013-11-15T14:00:00`

▼ add-days-to-dateTime [altova:]

```
altova:add-days-to-dateTime(DateTime as xs:dateTime, Days as xs:integer) as xs:dateTime XP3.1 XQ3.1
```

Adds a duration in days to an `xs:dateTime` (see examples below). The second argument is the number of days to be added to the `xs:dateTime` supplied as the first argument. The result is of type `xs:dateTime`.

▢ Examples

- `altova:add-days-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), 10)`  
returns `2014-01-25T14:00:00`
- `altova:add-days-to-dateTime(xs:dateTime("2014-01-15T14:00:00"), -8)`  
returns `2014-01-07T14:00:00`

▼ add-hours-to-dateTime [altova:]

```
altova:add-hours-to-dateTime(DateTime as xs:dateTime, Hours as xs:integer) as xs:dateTime XP3.1 XQ3.1
```

Adds a duration in hours to an `xs:dateTime` (see examples below). The second argument is the number of hours to be added to the `xs:dateTime` supplied as the first argument. The result is of type `xs:dateTime`.

▢ Examples

- `altova:add-hours-to-dateTime(xs:dateTime("2014-01-15T13:00:00"), 10)`  
returns `2014-01-15T23:00:00`
- `altova:add-hours-to-dateTime(xs:dateTime("2014-01-15T13:00:00"), -8)`  
returns `2014-01-15T05:00:00`

▼ add-minutes-to-dateTime [altova:]

```
altova:add-minutes-to-dateTime(DateTime as xs:dateTime, Minutes as xs:integer) as xs:dateTime XP3.1 XQ3.1
```

Adds a duration in minutes to an `xs:dateTime` (see examples below). The second argument is the number of minutes to be added to the `xs:dateTime` supplied as the first argument. The result is of type `xs:dateTime`.

▢ Examples

- `altova:add-minutes-to-dateTime(xs:dateTime("2014-01-15T14:10:00"), 45)`  
returns `2014-01-15T14:55:00`



- **altova:add-minutes-to-dateTime**(xs:dateTime("2014-01-15T14:10:00"), -5)  
returns 2014-01-15T14:05:00

▼ add-seconds-to-dateTime [altova:]

**altova:add-seconds-to-dateTime**(DateTime as xs:dateTime, Seconds as xs:integer) as xs:dateTime **XP3.1 XQ3.1**

Adds a duration in seconds to an xs:dateTime (see examples below). The second argument is the number of seconds to be added to the xs:dateTime supplied as the first argument. The result is of type xs:dateTime.

☞ Examples

- **altova:add-seconds-to-dateTime**(xs:dateTime("2014-01-15T14:00:10"), 20)  
returns 2014-01-15T14:00:30
- **altova:add-seconds-to-dateTime**(xs:dateTime("2014-01-15T14:00:10"), -5)  
returns 2014-01-15T14:00:05

[\[ Top \]](#)

### Add a duration to xs:date **XP3.1 XQ3.1**

These functions add a duration to xs:date and return xs:date. The xs:date type has a format of CCYY-MM-DD.

▼ add-years-to-date [altova:]

**altova:add-years-to-date**(Date as xs:date, Years as xs:integer) as xs:date **XP3.1 XQ3.1**

Adds a duration in years to a date. The second argument is the number of years to be added to the xs:date supplied as the first argument. The result is of type xs:date.

☞ Examples

- **altova:add-years-to-date**(xs:date("2014-01-15"), 10) returns 2024-01-15
- **altova:add-years-to-date**(xs:date("2014-01-15"), -4) returns 2010-01-15

▼ add-months-to-date [altova:]

**altova:add-months-to-date**(Date as xs:date, Months as xs:integer) as xs:date **XP3.1 XQ3.1**

Adds a duration in months to a date. The second argument is the number of months to be added to the xs:date supplied as the first argument. The result is of type xs:date.

☞ Examples

- **altova:add-months-to-date**(xs:date("2014-01-15"), 10) returns 2014-11-15
- **altova:add-months-to-date**(xs:date("2014-01-15"), -2) returns 2013-11-15

▼ add-days-to-date [altova:]



**altova:add-days-to-date**(Date as xs:date, Days as xs:integer) as xs:date XP3.1 XQ3.1

Adds a duration in days to a date. The second argument is the number of days to be added to the xs:date supplied as the first argument. The result is of type xs:date.

▣ Examples

- **altova:add-days-to-date**(xs:date("2014-01-15"), 10) returns 2014-01-25
- **altova:add-days-to-date**(xs:date("2014-01-15"), -8) returns 2014-01-07

[\[ Top \]](#)

## Format and retrieve durations XP3.1 XQ3.1

These functions add a duration to xs:date and return xs:date. The xs:date type has a format of CCYY-MM-DD.

### ▼ format-duration [altova:]

**altova:format-duration**(Duration as xs:duration, Picture as xs:string) as xs:string XP3.1 XQ3.1

Formats a duration, which is submitted as the first argument, according to a picture string submitted as the second argument. The output is a text string formatted according to the picture string.

▣ Examples

- **altova:format-duration**(xs:duration("P2DT2H53M11.7S"), "Days:[D01] Hours:[H01] Minutes:[m01] Seconds:[s01] Fractions:[f0]") returns "Days:02 Hours:02 Minutes:53 Seconds:11 Fractions:7"
- **altova:format-duration**(xs:duration("P3M2DT2H53M11.7S"), "Months:[M01] Days:[D01] Hours:[H01] Minutes:[m01]") returns "Months:03 Days:02 Hours:02 Minutes:53"

### ▼ parse-duration [altova:]

**altova:parse-duration**(InputString as xs:string, Picture as xs:string) as xs:duration XP3.1 XQ3.1

Takes a patterned string as the first argument, and a picture string as the second argument. The input string is parsed on the basis of the picture string, and an xs:duration is returned.

▣ Examples

- **altova:parse-duration**("Days:02 Hours:02 Minutes:53 Seconds:11 Fractions:7", "Days:[D01] Hours:[H01] Minutes:[m01] Seconds:[s01] Fractions:[f0]") returns "P2DT2H53M11.7S"
- **altova:parse-duration**("Months:03 Days:02 Hours:02 Minutes:53 Seconds:11 Fractions:7", "Months:[M01] Days:[D01] Hours:[H01] Minutes:[m01]") returns "P3M2DT2H53M"

[\[ Top \]](#)



### Add a duration to `xs:time` **XP3.1 XQ3.1**

These functions add a duration to `xs:time` and return `xs:time`. The `xs:time` type has a lexical form of `hh:mm:ss.sss`. An optional time zone may be suffixed. The letter `z` indicates Coordinated Universal Time (UTC). All other time zones are represented by their difference from UTC in the format `+hh:mm`, or `-hh:mm`. If no time zone value is present, it is considered unknown; it is not assumed to be UTC.

#### ▼ add-hours-to-time [altova:]

**altova:add-hours-to-time**(*Time* as `xs:time`, *Hours* as `xs:integer`) as `xs:time`  
**XP3.1 XQ3.1**

Adds a duration in hours to a time. The second argument is the number of hours to be added to the `xs:time` supplied as the first argument. The result is of type `xs:time`.

##### ▢ Examples

- **altova:add-hours-to-time**(`xs:time("11:00:00")`, 10) returns `21:00:00`
- **altova:add-hours-to-time**(`xs:time("11:00:00")`, -7) returns `04:00:00`

#### ▼ add-minutes-to-time [altova:]

**altova:add-minutes-to-time**(*Time* as `xs:time`, *Minutes* as `xs:integer`) as `xs:time`  
**XP3.1 XQ3.1**

Adds a duration in minutes to a time. The second argument is the number of minutes to be added to the `xs:time` supplied as the first argument. The result is of type `xs:time`.

##### ▢ Examples

- **altova:add-minutes-to-time**(`xs:time("14:10:00")`, 45) returns `14:55:00`
- **altova:add-minutes-to-time**(`xs:time("14:10:00")`, -5) returns `14:05:00`

#### ▼ add-seconds-to-time [altova:]

**altova:add-seconds-to-time**(*Time* as `xs:time`, *Seconds* as `xs:integer`) as `xs:time`  
**XP3.1 XQ3.1**

Adds a duration in seconds to a time. The second argument is the number of seconds to be added to the `xs:time` supplied as the first argument. The result is of type `xs:time`. The Seconds component can be in the range of 0 to 59.999.

##### ▢ Examples

- **altova:add-seconds-to-time**(`xs:time("14:00:00")`, 20) returns `14:00:20`
- **altova:add-seconds-to-time**(`xs:time("14:00:00")`, 20.895) returns `14:00:20.895`

[\[ Top \]](#)

### Remove the timezone part from date/time datatypes **XP3.1 XQ3.1**

These functions remove the timezone from the current `xs:dateTime`, `xs:date`, or `xs:time` values,



respectively. Note that the difference between `xs:dateTime` and `xs:dateTimeStamp` is that in the case of the latter the timezone part is required (while it is optional in the case of the former). So the format of an `xs:dateTimeStamp` value is: `CCYY-MM-DDThh:mm:ss.sss±hh:mm`. or `CCYY-MM-DDThh:mm:ss.sssZ`. If the date and time is read from the system clock as `xs:dateTimeStamp`, the `current-dateTime-no-TZ()` function can be used to remove the timezone if so required.

▼ `current-dateTime-no-TZ` [altova:]

**`altova:current-dateTime-no-TZ()`** as **`xs:dateTime`** **XP3.1 XQ3.1**

This function takes no argument. It removes the timezone part of `current-dateTime()` (which is the current date-and-time according to the system clock) and returns an `xs:dateTime` value.

▢ Examples

If the current `dateTime` is `2014-01-15T14:00:00+01:00`:

- **`altova:current-dateTime-no-TZ()`** returns `2014-01-15T14:00:00`

▼ `current-date-no-TZ` [altova:]

**`altova:current-date-no-TZ()`** as **`xs:date`** **XP3.1 XQ3.1**

This function takes no argument. It removes the timezone part of `current-date()` (which is the current date according to the system clock) and returns an `xs:date` value.

▢ Examples

If the current date is `2014-01-15+01:00`:

- **`altova:current-date-no-TZ()`** returns `2014-01-15`

▼ `current-time-no-TZ` [altova:]

**`altova:current-time-no-TZ()`** as **`xs:time`** **XP3.1 XQ3.1**

This function takes no argument. It removes the timezone part of `current-time()` (which is the current time according to the system clock) and returns an `xs:time` value.

▢ Examples

If the current time is `14:00:00+01:00`:

- **`altova:current-time-no-TZ()`** returns `14:00:00`

[\[ Top \]](#)

**Return the weekday from `xs:dateTime` OR `xs:date`** **XP3.1 XQ3.1**

These functions return the weekday (as an integer) from `xs:dateTime` or `xs:date`. The days of the week are numbered (using the American format) from 1 to 7, with Sunday=1. In the European format, the week starts with Monday (=1). The American format, where Sunday=1, can be set by using the integer 0 where an integer is accepted to indicate the format.



▼ weekday-from-dateTime [altova:]

**altova:weekday-from-dateTime**(DateTime as xs:dateTime) as xs:integer XP3.1 XQ3.1

Takes a date-with-time as its single argument and returns the day of the week of this date as an integer. The weekdays are numbered starting with Sunday=1. If the European format is required (where Monday=1), use the other signature of this function (see *next signature below*).

▢ Examples

- **altova:weekday-from-dateTime**(xs:dateTime("2014-02-03T09:00:00")) returns 2, which would indicate a Monday.

**altova:weekday-from-dateTime**(DateTime as xs:dateTime, Format as xs:integer) as xs:integer XP3.1 XQ3.1

Takes a date-with-time as its first argument and returns the day of the week of this date as an integer. The weekdays are numbered starting with Monday=1. If the second (integer) argument is 0, then the weekdays are numbered 1 to 7 starting with Sunday=1. If the second argument is an integer other than 0, then Monday=1. If there is no second argument, the function is read as having the other signature of this function (see *previous signature*).

▢ Examples

- **altova:weekday-from-dateTime**(xs:dateTime("2014-02-03T09:00:00"), 1) returns 1, which would indicate a Monday
- **altova:weekday-from-dateTime**(xs:dateTime("2014-02-03T09:00:00"), 4) returns 1, which would indicate a Monday
- **altova:weekday-from-dateTime**(xs:dateTime("2014-02-03T09:00:00"), 0) returns 2, which would indicate a Monday.

▼ weekday-from-date [altova:]

**altova:weekday-from-date**(Date as xs:date) as xs:integer XP3.1 XQ3.1

Takes a date as its single argument and returns the day of the week of this date as an integer. The weekdays are numbered starting with Sunday=1. If the European format is required (where Monday=1), use the other signature of this function (see *next signature below*).

▢ Examples

- **altova:weekday-from-date**(xs:date("2014-02-03+01:00")) returns 2, which would indicate a Monday.

**altova:weekday-from-date**(Date as xs:date, Format as xs:integer) as xs:integer XP3.1 XQ3.1

Takes a date as its first argument and returns the day of the week of this date as an integer. The weekdays are numbered starting with Monday=1. If the second (Format) argument is 0, then the weekdays are numbered 1 to 7 starting with Sunday=1. If the second argument is an integer other than 0, then Monday=1. If there is no second argument, the function is read as having the other signature of this function (see *previous signature*).

▢ Examples

- **altova:weekday-from-date**(xs:date("2014-02-03"), 1) returns 1, which would indicate a Monday
- **altova:weekday-from-date**(xs:date("2014-02-03"), 4) returns 1, which would indicate a Monday



- **altova:weekday-from-date**(`xs:date("2014-02-03")`, 0) returns 2, which would indicate a Monday.

[\[ Top \]](#)

### Return the week number from `xs:dateTime` or `xs:date` XP2 XQ1 XP3.1 XQ3.1

These functions return the week number (as an integer) from `xs:dateTime` or `xs:date`. Week-numbering is available in the US, ISO/European, and Islamic calendar formats. Week-numbering is different in these calendar formats because the week is considered to start on different days (on Sunday in the US format, Monday in the ISO/European format, and Saturday in the Islamic format).

#### ▼ weeknumber-from-date [altova:]

```
altova:weeknumber-from-date(Date as xs:date, Calendar as xs:integer) as xs:integer XP2 XQ1 XP3.1 XQ3.1
```

Returns the week number of the submitted **Date** argument as an integer. The second argument (**calendar**) specifies the calendar system to follow.

Supported **calendar** values are:

- 0 = US calendar (*week starts Sunday*)
- 1 = ISO standard, European calendar (*week starts Monday*)
- 2 = Islamic calendar (*week starts Saturday*)

Default is 0.

#### ▢ Examples

- **altova:weeknumber-from-date**(`xs:date("2014-03-23")`, 0) returns 13
- **altova:weeknumber-from-date**(`xs:date("2014-03-23")`, 1) returns 12
- **altova:weeknumber-from-date**(`xs:date("2014-03-23")`, 2) returns 13
- **altova:weeknumber-from-date**(`xs:date("2014-03-23")`) returns 13

The day of the date in the examples above (2014-03-23) is Sunday. So the US and Islamic calendars are one week ahead of the European calendar on this day.

#### ▼ weeknumber-from-dateTime [altova:]

```
altova:weeknumber-from-dateTime(DateTime as xs:dateTime, Calendar as xs:integer) as xs:integer XP2 XQ1 XP3.1 XQ3.1
```

Returns the week number of the submitted **DateTime** argument as an integer. The second argument (**calendar**) specifies the calendar system to follow.

Supported **calendar** values are:

- 0 = US calendar (*week starts Sunday*)
- 1 = ISO standard, European calendar (*week starts Monday*)
- 2 = Islamic calendar (*week starts Saturday*)



Default is 0.

#### Examples

- `altova:weeknumber-from-dateTime(xs:dateTime("2014-03-23T00:00:00"), 0)`  
returns 13
- `altova:weeknumber-from-dateTime(xs:dateTime("2014-03-23T00:00:00"), 1)`  
returns 12
- `altova:weeknumber-from-dateTime(xs:dateTime("2014-03-23T00:00:00"), 2)`  
returns 13
- `altova:weeknumber-from-dateTime(xs:dateTime("2014-03-23T00:00:00") )`  
returns 13

The day of the dateTime in the examples above (2014-03-23T00:00:00) is Sunday. So the US and Islamic calendars are one week ahead of the European calendar on this day.

[\[ Top \]](#)

### Build date, time, and duration datatypes from their lexical components XP3.1 XQ3.1

The functions take the lexical components of the `xs:date`, `xs:time`, or `xs:duration` datatype as input arguments and combine them to build the respective datatype.

#### ▼ build-date [altova:]

`altova:build-date(Year as xs:integer, Month as xs:integer, Date as xs:integer) as xs:date` XP3.1 XQ3.1

The first, second, and third arguments are, respectively, the year, month, and date. They are combined to build a value of `xs:date` type. The values of the integers must be within the correct range of that particular date part. For example, the second argument (for the month part) should not be greater than 12.

#### Examples

- `altova:build-date(2014, 2, 03)` returns 2014-02-03

#### ▼ build-time [altova:]

`altova:build-time(Hours as xs:integer, Minutes as xs:integer, Seconds as xs:integer) as xs:time` XP3.1 XQ3.1

The first, second, and third arguments are, respectively, the hour (0 to 23), minutes (0 to 59), and seconds (0 to 59) values. They are combined to build a value of `xs:time` type. The values of the integers must be within the correct range of that particular time part. For example, the second (Minutes) argument should not be greater than 59. To add a timezone part to the value, use the other signature of this function (see next signature).

#### Examples

- `altova:build-time(23, 4, 57)` returns 23:04:57



```
altova:build-time(Hours as xs:integer, Minutes as xs:integer, Seconds as xs:integer, TimeZone as xs:string) as xs:time XP3.1 XQ3.1
```

The first, second, and third arguments are, respectively, the hour (0 to 23), minutes (0 to 59), and seconds (0 to 59) values. The fourth argument is a string that provides the timezone part of the value. The four arguments are combined to build a value of `xs:time` type. The values of the integers must be within the correct range of that particular time part. For example, the second (`Minutes`) argument should not be greater than 59.

▣ Examples

- `altova:build-time(23, 4, 57, '+1')` returns `23:04:57+01:00`

▼ **build-duration** [altova:]

```
altova:build-duration(Years as xs:integer, Months as xs:integer) as xs:yearMonthDuration XP3.1 XQ3.1
```

Takes two arguments to build a value of type `xs:yearMonthDuration`. The first argument provides the `Years` part of the duration value, while the second argument provides the `Months` part. If the second (`Months`) argument is greater than or equal to 12, then the integer is divided by 12; the quotient is added to the first argument to provide the `Years` part of the duration value while the remainder (of the division) provides the `Months` part. To build a duration of type `xs:dayTimeDuration`, see the next signature.

▣ Examples

- `altova:build-duration(2, 10)` returns `P2Y10M`
- `altova:build-duration(14, 27)` returns `P16Y3M`
- `altova:build-duration(2, 24)` returns `P4Y`

```
altova:build-duration(Days as xs:integer, Hours as xs:integer, Minutes as xs:integer, Seconds as xs:integer) as xs:dayTimeDuration XP3.1 XQ3.1
```

Takes four arguments and combines them to build a value of type `xs:dayTimeDuration`. The first argument provides the `Days` part of the duration value, the second, third, and fourth arguments provide, respectively, the `Hours`, `Minutes`, and `Seconds` parts of the duration value. Each of the three `Time` arguments is converted to an equivalent value in terms of the next higher unit and the result is used for calculation of the total duration value. For example, 72 seconds is converted to `1M+12S` (1 minute and 12 seconds), and this value is used for calculation of the total duration value. To build a duration of type `xs:yearMonthDuration`, see the previous signature.

▣ Examples

- `altova:build-duration(2, 10, 3, 56)` returns `P2DT10H3M56S`
- `altova:build-duration(1, 0, 100, 0)` returns `P1DT1H40M`
- `altova:build-duration(1, 0, 0, 3600)` returns `P1DT1H`

[\[ Top \]](#)

## Construct date, dateTime, and time datatypes from string input XP2 XQ1 XP3.1 XQ3.1

These functions take strings as arguments and construct `xs:date`, `xs:dateTime`, or `xs:time` datatypes. The string is analyzed for components of the datatype based on a submitted pattern



argument.

▼ `parse-date` [altova:]

`altova:parse-date(Date as xs:string, DatePattern as xs:string) as xs:date`  
**XP2 XQ1 XP3.1 XQ3.1**

Returns the input string `Date` as an `xs:date` value. The second argument `DatePattern` specifies the pattern (sequence of components) of the input string. `DatePattern` is described with the component specifiers listed below and with component separators that can be any character. See the examples below.

<b>D</b>	Date
<b>M</b>	Month
<b>Y</b>	Year

The pattern in `DatePattern` must match the pattern in `Date`. Since the output is of type `xs:date`, the output will always have the lexical format `YYYY-MM-DD`.

▢ Examples

- `altova:parse-date(xs:string("09-12-2014"), "[D]-[M]-[Y]")` returns `2014-12-09`
- `altova:parse-date(xs:string("09-12-2014"), "[M]-[D]-[Y]")` returns `2014-09-12`
- `altova:parse-date("06/03/2014", "[M]/[D]/[Y]")` returns `2014-06-03`
- `altova:parse-date("06 03 2014", "[M] [D] [Y]")` returns `2014-06-03`
- `altova:parse-date("6 3 2014", "[M] [D] [Y]")` returns `2014-06-03`

▼ `parse-dateTime` [altova:]

`altova:parse-dateTime(DateTime as xs:string, DateTimePattern as xs:string) as xs:dateTime`  
**XP2 XQ1 XP3.1 XQ3.1**

Returns the input string `DateTime` as an `xs:dateTime` value. The second argument `DateTimePattern` specifies the pattern (sequence of components) of the input string. `DateTimePattern` is described with the component specifiers listed below and with component separators that can be any character. See the examples below.

<b>D</b>	Date
<b>M</b>	Month
<b>Y</b>	Year
<b>H</b>	Hour
<b>m</b>	minutes
<b>s</b>	seconds

The pattern in `DateTimePattern` must match the pattern in `DateTime`. Since the output is of type `xs:dateTime`, the output will always have the lexical format `YYYY-MM-DDTHH:mm:ss`.

▢ Examples

- `altova:parse-dateTime(xs:string("09-12-2014 13:56:24"), "[M]-[D]-[Y][H]:[m]:[s]")` returns `2014-09-12T13:56:24`
- `altova:parse-dateTime("time=13:56:24; date=09-12-2014", "time=[H]:[m]:[s]; date=[M]-[D]-[Y]")` returns `2014-09-12T13:56:24`



```
[s]; date=[D]-[M]-[Y]") returns 2014-12-09T13:56:24
```

#### ▼ parse-time [altova:]

**altova:parse-time**(**Time** as *xs:string*, **TimePattern** as *xs:string*) as *xs:time* **XP2 XQ1 XP3.1 XQ3.1**

Returns the input string **Time** as an *xs:time* value. The second argument **TimePattern** specifies the pattern (sequence of components) of the input string. **TimePattern** is described with the component specifiers listed below and with component separators that can be any character. See the examples below.

<b>H</b>	Hour
<b>m</b>	minutes
<b>s</b>	seconds

The pattern in **TimePattern** must match the pattern in **Time**. Since the output is of type *xs:time*, the output will always have the lexical format **HH:mm:ss**.

#### ▣ Examples

- **altova:parse-time**(*xs:string*("13:56:24"), "[H]:[m]:[s]") returns 13:56:24
- **altova:parse-time**("13-56-24", "[H]-[m]") returns 13:56:00
- **altova:parse-time**("time=13h56m24s", "time=[H]h[m]m[s]s") returns 13:56:24
- **altova:parse-time**("time=24s56m13h", "time=[s]s[m]m[H]h") returns 13:56:24

[\[ Top \]](#)

### Age-related functions **XP3.1 XQ3.1**

These functions return the age as calculated (i) between one input argument date and the current date, or (ii) between two input argument dates. The **altova:age** function returns the age in terms of years, the **altova:age-details** function returns the age as a sequence of three integers giving the years, months, and days of the age.

#### ▼ age [altova:]

**altova:age**(**StartDate** as *xs:date*) as *xs:integer* **XP3.1 XQ3.1**

Returns an integer that is the age *in years* of some object, counting from a start-date submitted as the argument and ending with the current date (taken from the system clock). If the input argument is a date anything greater than or equal to one year in the future, the return value will be negative.

#### ▣ Examples

If the current date is 2014-01-15:

- **altova:age**(*xs:date*("2013-01-15")) returns 1



- **altova:age**(xs:date("2013-01-16")) returns 0
- **altova:age**(xs:date("2015-01-15")) returns -1
- **altova:age**(xs:date("2015-01-14")) returns 0

**altova:age**(StartDate as xs:date, EndDate as xs:date) as xs:integer XP3.1 XQ3.1

Returns an integer that is the age *in years* of some object, counting from a start-date that is submitted as the first argument up to an end-date that is the second argument. The return value will be negative if the first argument is one year or more later than the second argument.

▣ Examples

If the current date is 2014-01-15:

- **altova:age**(xs:date("2000-01-15"), xs:date("2010-01-15")) returns 10
- **altova:age**(xs:date("2000-01-15"), current-date()) returns 14 if the current date is 2014-01-15
- **altova:age**(xs:date("2014-01-15"), xs:date("2010-01-15")) returns -4

▼ age-details [altova:]

**altova:age-details**(InputDate as xs:date) as (xs:integer)\* XP3.1 XQ3.1

Returns three integers that are, respectively, the years, months, and days between the date that is submitted as the argument and the current date (taken from the system clock). The sum of the returned *years+months+days* together gives the total time difference between the two dates (the input date and the current date). The input date may have a value earlier or later than the current date, but whether the input date is earlier or later is not indicated by the sign of the return values; the return values are always positive.

▣ Examples

If the current date is 2014-01-15:

- **altova:age-details**(xs:date("2014-01-16")) returns (0 0 1)
- **altova:age-details**(xs:date("2014-01-14")) returns (0 0 1)
- **altova:age-details**(xs:date("2013-01-16")) returns (1 0 1)
- **altova:age-details**(current-date()) returns (0 0 0)

**altova:age-details**(Date-1 as xs:date, Date-2 as xs:date) as (xs:integer)\* XP3.1 XQ3.1

Returns three integers that are, respectively, the years, months, and days between the two argument dates. The sum of the returned *years+months+days* together gives the total time difference between the two input dates; it does not matter whether the earlier or later of the two dates is submitted as the first argument. The return values do not indicate whether the input date occurs earlier or later than the current date. Return values are always positive.

▣ Examples

- **altova:age-details**(xs:date("2014-01-16"), xs:date("2014-01-15")) returns (0 0 1)
- **altova:age-details**(xs:date("2014-01-15"), xs:date("2014-01-16")) returns (0 0 1)



[\[ Top \]](#)

### *XPath/XQuery Functions: Geolocation*

The following geolocation XPath/XQuery extension functions are supported in the current version of MapForce and can be used in (i) XPath expressions in an XSLT context, or (ii) XQuery expressions in an XQuery document.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, <http://www.altova.com/xslt-extensions>, and are indicated in this section with the prefix **altova:**, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

<i>XPath functions (used in XPath expressions in XSLT):</i>	<b>XP1</b> <b>XP2</b> <b>XP3.1</b>
<i>XSLT functions (used in XPath expressions in XSLT):</i>	<b>XSLT1</b> <b>XSLT2</b> <b>XSLT3</b>
<i>XQuery functions (used in XQuery expressions in XQuery):</i>	<b>XQ1</b> <b>XQ3.1</b>

#### ▼ **parse-geolocation** [altova:]

**altova:parse-geolocation**(**GeolocationInputString** as *xs:string*) as *xs:decimal+*  
**XP3.1** **XQ3.1**

Parses the supplied **GeolocationInputString** argument and returns the geolocation's latitude and longitude (in that order) as a sequence two *xs:decimal* items. The formats in which the geolocation input string can be supplied are listed below.

**Note:** The [image-exif-data](#) function and the Exif metadata's [@Geolocation](#) attribute can be used to supply the geolocation input string (see *example below*).

#### ☐ Examples

- **altova:parse-geolocation**("33.33 -22.22") returns the sequence of two *xs:decimals* (33.33, 22.22)
- **altova:parse-geolocation**("48°51'29.6"N 24°17'40.2"W") returns the sequence of two *xs:decimals* (48.858222222222, 24.2945)
- **altova:parse-geolocation**("48°51'29.6"N 24°17'40.2"W") returns the sequence of two *xs:decimals* (48.858222222222, 24.2945)
- **altova:parse-geolocation**( **image-exif-data**(//MyImages/Image20141130.01)/**@Geolocation** ) returns a sequence of two *xs:decimals*

#### ☐ Geolocation input string formats:

The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).



**Note:** If single quotes or double quotes are used to delimit the input string argument, this will create a mismatch with the single quotes or double quotes that are used, respectively, to indicate minute-values and second-values. In such cases, the quotes that are used for indicating minute-values and second-values must be escaped by doubling them. In the examples in this section, quotes used to delimit the input string are highlighted in yellow (") while unit indicators that are escaped are highlighted in blue (").

- Degrees, minutes, decimal seconds, with suffixed orientation (N/S, W/E)  
D°M'S.SS"N/S D°M'S.SS"W/E  
*Example:* 33°55'11.11"N 22°44'55.25"W
- Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D°M'S.SS" +/-D°M'S.SS"  
*Example:* 33°55'11.11" -22°44'55.25"
- Degrees, decimal minutes, with suffixed orientation (N/S, W/E)  
D°M.MM'N/S D°M.MM'W/E  
*Example:* 33°55.55'N 22°44.44'W
- Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D°M.MM' +/-D°M.MM'  
*Example:* +33°55.55' -22°44.44'
- Decimal degrees, with suffixed orientation (N/S, W/E)  
D.DDN/S D.DDW/E  
*Example:* 33.33N 22.22W
- Decimal degrees, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D.DD +/-D.DD  
*Example:* 33.33 -22.22

Examples of format-combinations:

33.33N -22°44'55.25"  
33.33 22°44'55.25"W  
33.33 22.45

☐ Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute **Geolocation** from standard Exif metadata tags. **Geolocation** is a concatenation of four Exif tags: GPSLatitude, GPSLatitudeRef, GPSLongitude, GPSLongitudeRef, with units added (see table below).

GPSLatitude	GPSLatitudeRef	GPSLongitude	GPSLongitudeRef	Geolocation
33 51 21.91	S	151 13 11.73	E	33°51'21.91"S 151° 13'11.73"E



## ▼ geolocation-distance-km [altova:]

```
altova:geolocation-distance-km(GeolocationInputString-1 as xs:string,
GeolocationInputString-2 as xs:string) as xs:decimal XP3.1 XQ3.1
```

Calculates the distance between two geolocations in kilometers. The formats in which the geolocation input string can be supplied are listed below. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

**Note:** The [image-exif-data](#) function and the Exif metadata's [@Geolocation](#) attribute can be used to supply geolocation input strings.

☐ Examples

- `altova:geolocation-distance-km("33.33 -22.22", "48°51'29.6"N 24°17'40.2"W")` returns the `xs:decimal` `4183.08132372392`

☐ Geolocation input string formats:

The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

**Note:** If single quotes or double quotes are used to delimit the input string argument, this will create a mismatch with the single quotes or double quotes that are used, respectively, to indicate minute-values and second-values. In such cases, the quotes that are used for indicating minute-values and second-values must be escaped by doubling them. In the examples in this section, quotes used to delimit the input string are highlighted in yellow (") while unit indicators that are escaped are highlighted in blue (").

- Degrees, minutes, decimal seconds, with suffixed orientation (N/S, W/E)  
`D°M'S.SS"N/S D°M'S.SS"W/E`  
Example: `33°55'11.11"N 22°44'55.25"W`
- Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/W) is optional  
`+/-D°M'S.SS" +/-D°M'S.SS"`  
Example: `33°55'11.11" -22°44'55.25"`
- Degrees, decimal minutes, with suffixed orientation (N/S, W/E)  
`D°M.MM"N/S D°M.MM"W/E`  
Example: `33°55.55'N 22°44.44'W`
- Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/W) is optional  
`+/-D°M.MM' +/-D°M.MM'`  
Example: `+33°55.55' -22°44.44'`



- Decimal degrees, with suffixed orientation (N/S, W/E)  
D.DDN/S D.DDW/E  
*Example:* 33.33N 22.22W
- Decimal degrees, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D.DD +/-D.DD  
*Example:* 33.33 -22.22

Examples of format-combinations:

33.33N -22°44'55.25"  
33.33 22°44'55.25"W  
33.33 22.45

☐ Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute **Geolocation** from standard Exif metadata tags. **Geolocation** is a concatenation of four Exif tags: GPSLatitude, GPSLatitudeRef, GPSLongitude, GPSLongitudeRef, with units added (see table below).

GPSLatitude	GPSLatitudeRef	GPSLongitude	GPSLongitudeRef	Geolocation
33 51 21.91	S	151 13 11.73	E	33°51'21.91"S 151° 13'11.73"E

▼ geolocation-distance-mi [altova:]

**altova:geolocation-distance-mi**(**GeolocationInputString-1** as **xs:string**, **GeolocationInputString-2** as **xs:string**) as **xs:decimal** **XP3.1 XQ3.1**

Calculates the distance between two geolocations in miles. The formats in which a geolocation input string can be supplied are listed below. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

**Note:** The [image-exif-data](#) function and the Exif metadata's [@Geolocation](#) attribute can be used to supply geolocation input strings.

☐ Examples

- **altova:geolocation-distance-mi**("33.33 -22.22", "48°51'29.6"N 24°17'40.2"E") returns the **xs:decimal** 2599.40652340653

☐ Geolocation input string formats:

The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).



**Note:** If single quotes or double quotes are used to delimit the input string argument, this will create a mismatch with the single quotes or double quotes that are used, respectively, to indicate minute-values and second-values. In such cases, the quotes that are used for indicating minute-values and second-values must be escaped by doubling them. In the examples in this section, quotes used to delimit the input string are highlighted in yellow (") while unit indicators that are escaped are highlighted in blue ("").

- Degrees, minutes, decimal seconds, with suffixed orientation (N/S, W/E)  
D°M'S.SS"N/S D°M'S.SS"W/E  
*Example:* 33°55'11.11"N 22°44'55.25"W
- Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D°M'S.SS" +/-D°M'S.SS"  
*Example:* 33°55'11.11" -22°44'55.25"
- Degrees, decimal minutes, with suffixed orientation (N/S, W/E)  
D°M.MM"N/S D°M.MM"W/E  
*Example:* 33°55.55'N 22°44.44'W
- Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D°M.MM' +/-D°M.MM'  
*Example:* +33°55.55' -22°44.44'
- Decimal degrees, with suffixed orientation (N/S, W/E)  
D.DDN/S D.DDW/E  
*Example:* 33.33N 22.22W
- Decimal degrees, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D.DD +/-D.DD  
*Example:* 33.33 -22.22

Examples of format-combinations:

33.33N -22°44'55.25"  
33.33 22°44'55.25"W  
33.33 22.45

☐ Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute **Geolocation** from standard Exif metadata tags. **Geolocation** is a concatenation of four Exif tags: GPSLatitude, GPSLatitudeRef, GPSLongitude, GPSLongitudeRef, with units added (see table below).

GPSLatitude	GPSLatitudeRef	GPSLongitude	GPSLongitudeRef	Geolocation
33 51 21.91	S	151 13 11.73	E	33°51'21.91"S 151° 13'11.73"E



## ▼ geolocation-within-polygon [altova:]

```
altova:geolocation-within-polygon(Geolocation as xs:string, ((PolygonPoint
as xs:string)+)) as xs:boolean XP3.1 XQ3.1
```

Determines whether `Geolocation` (the first argument) is within the polygonal area described by the `PolygonPoint` arguments. If the `PolygonPoint` arguments do not form a closed figure (formed when the first point and the last point are the same), then the first point is implicitly added as the last point in order to close the figure. All the arguments (`Geolocation` and `PolygonPoint`+) are given by geolocation input strings (*formats listed below*). If the `Geolocation` argument is within the polygonal area, then the function returns `true()`; otherwise it returns `false()`. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

**Note:** The [image-exif-data](#) function and the Exif metadata's `@Geolocation` attribute can be used to supply geolocation input strings.

☐ Examples

- `altova:geolocation-within-polygon("33 -22", ("58 -32", "-78 -55", "48 24", "58 -32"))` returns `true()`
- `altova:geolocation-within-polygon("33 -22", ("58 -32", "-78 -55", "48 24"))` returns `true()`
- `altova:geolocation-within-polygon("33 -22", ("58 -32", "-78 -55", "48°51'29.6"N 24°17'40.2"W"))` returns `true()`

☐ Geolocation input string formats:

The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

**Note:** If single quotes or double quotes are used to delimit the input string argument, this will create a mismatch with the single quotes or double quotes that are used, respectively, to indicate minute-values and second-values. In such cases, the quotes that are used for indicating minute-values and second-values must be escaped by doubling them. In the examples in this section, quotes used to delimit the input string are highlighted in yellow (") while unit indicators that are escaped are highlighted in blue (").

- Degrees, minutes, decimal seconds, with suffixed orientation (N/S, W/E)  
`D°M'S.SS"N/S D°M'S.SS"W/E`  
Example: 33°55'11.11"N 22°44'55.25"W
- Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/W) is optional  
`+/-D°M'S.SS" +/-D°M'S.SS"`  
Example: 33°55'11.11" -22°44'55.25"



- Degrees, decimal minutes, with suffixed orientation (N/S, W/E)  
D°M.MM'N/S D°M.MM'W/E  
*Example:* 33°55.55'N 22°44.44'W
- Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D°M.MM' +/-D°M.MM'  
*Example:* +33°55.55' -22°44.44'
- Decimal degrees, with suffixed orientation (N/S, W/E)  
D.DDN/S D.DDW/E  
*Example:* 33.33N 22.22W
- Decimal degrees, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D.DD +/-D.DD  
*Example:* 33.33 -22.22

Examples of format-combinations:

33.33N -22°44'55.25"

33.33 22°44'55.25"W

33.33 22.45

Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute `Geolocation` from standard Exif metadata tags. `Geolocation` is a concatenation of four Exif tags: `GPSLatitude`, `GPSLatitudeRef`, `GPSLongitude`, `GPSLongitudeRef`, with units added (see table below).

GPSPatitu de	GPSPatitu Ref	GPSLongitu de	GPSLongitude Ref	Geolocation
33 51 21.91	S	151 13 11.73	E	33°51'21.91"S 151° 13'11.73"E

▼ geolocation-within-rectangle [altova:]

`altova:geolocation-within-rectangle(Geolocation as xs:string, RectCorner-1 as xs:string, RectCorner-2 as xs:string) as xs:boolean XP3.1 XQ3.1`

Determines whether `Geolocation` (the first argument) is within the rectangle defined by the second and third arguments, `RectCorner-1` and `RectCorner-2`, which specify opposite corners of the rectangle. All the arguments (`Geolocation`, `RectCorner-1` and `RectCorner-2`) are given by geolocation input strings (*formats listed below*). If the `Geolocation` argument is within the rectangle, then the function returns `true()`; otherwise it returns `false()`. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

**Note:** The [image-exif-data](#) function and the Exif metadata's `@Geolocation` attribute can be used to supply geolocation input strings.



### Examples

- `altova:geolocation-within-rectangle("33 -22", "58 -32", "-48 24")`  
returns `true()`
- `altova:geolocation-within-rectangle("33 -22", "58 -32", "48 24")` returns  
`false()`
- `altova:geolocation-within-rectangle("33 -22", "58 -32", "48°51'29.6"S  
24°17'40.2"E")` returns `true()`

### Geolocation input string formats:

The geolocation input string must contain latitude and longitude (in that order) separated by whitespace. Each can be in any of the following formats. Combinations are allowed. So latitude can be in one format and longitude can be in another. Latitude values range from +90 to -90 (N to S). Longitude values range from +180 to -180 (E to W).

**Note:** If single quotes or double quotes are used to delimit the input string argument, this will create a mismatch with the single quotes or double quotes that are used, respectively, to indicate minute-values and second-values. In such cases, the quotes that are used for indicating minute-values and second-values must be escaped by doubling them. In the examples in this section, quotes used to delimit the input string are highlighted in yellow (") while unit indicators that are escaped are highlighted in blue (").

- Degrees, minutes, decimal seconds, with suffixed orientation (N/S, W/E)  
D°M'S.SS"N/S D°M'S.SS"W/E  
Example: 33°55'11.11"N 22°44'55.25"W
- Degrees, minutes, decimal seconds, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D°M'S.SS" +/-D°M'S.SS"  
Example: 33°55'11.11" -22°44'55.25"
- Degrees, decimal minutes, with suffixed orientation (N/S, W/E)  
D°M.MM"N/S D°M.MM"W/E  
Example: 33°55.55'N 22°44.44'W
- Degrees, decimal minutes, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D°M.MM' +/-D°M.MM'  
Example: +33°55.55' -22°44.44'
- Decimal degrees, with suffixed orientation (N/S, W/E)  
D.DDN/S D.DDW/E  
Example: 33.33N 22.22W
- Decimal degrees, with prefixed sign (+/-); the plus sign for (N/W) is optional  
+/-D.DD +/-D.DD  
Example: 33.33 -22.22

### Examples of format-combinations:

33.33N -22°44'55.25"



33.33 22°44'55.25"W

33.33 22.45

#### Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute **Geolocation** from standard Exif metadata tags. **Geolocation** is a concatenation of four Exif tags: GPSLatitude, GPSLatitudeRef, GPSLongitude, GPSLongitudeRef, with units added (see table below).

GPSLatitude	GPSLatitudeRef	GPSLongitude	GPSLongitudeRef	Geolocation
33 51 21.91	S	151 13 11.73	E	33°51'21.91"S 151° 13'11.73"E

[\[ Top \]](#)

### *XPath/XQuery Functions: Image-Related*

The following image-related XPath/XQuery extension functions are supported in the current version of MapForce and can be used in (i) XPath expressions in an XSLT context, or (ii) XQuery expressions in an XQuery document.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, <http://www.altova.com/xslt-extensions>, and are indicated in this section with the prefix **altova:**, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

XPath functions (used in XPath expressions in XSLT):	<b>XP1</b> <b>XP2</b> <b>XP3.1</b>
XSLT functions (used in XPath expressions in XSLT):	<b>XSLT1</b> <b>XSLT2</b> <b>XSLT3</b>
XQuery functions (used in XQuery expressions in XQuery):	<b>XQ1</b> <b>XQ3.1</b>

#### ▼ suggested-image-file-extension [altova:]

**altova:suggested-image-file-extension**(Base64String as string) as string?  
**XP3.1** **XQ3.1**

Takes the Base64 encoding of an image file as its argument and returns the file extension of the image as recorded in the Base64-encoding of the image. The returned value is a suggestion based on the image type information available in the encoding. If this information



is not available, then an empty string is returned. This function is useful if you wish to save a Base64 image as a file and wish to dynamically retrieve an appropriate file extension.

#### Examples

- `altova:suggested-image-file-extension(/MyImages/MobilePhone/Image20141130.01)` returns `'jpg'`
- `altova:suggested-image-file-extension($XML1/Staff/Person/@photo)` returns `''`

In the examples above, the nodes supplied as the argument of the function are assumed to contain a Base64-encoded image. The first example retrieves `jpg` as the file's type and extension. In the second example, the submitted Base64 encoding does not provide usable file extension information.

#### ▼ image-exif-data [altova:]

`altova:image-exif-data(Base64BinaryString as string) as element? XP3.1 XQ3.1`

Takes a Base64-encoded JPEG image as its argument and returns an element called `Exif` that contains the Exif metadata of the image. The Exif metadata is created as attribute-value pairs of the `Exif` element. The attribute names are the Exif data tags found in the Base64 encoding. The list of Exif-specification tags is given below. If a vendor-specific tag is present in the Exif data, this tag and its value will also be returned as an attribute-value pair. Additional to the standard Exif metadata tags (see *list below*), Altova-specific attribute-value pairs are also generated. These Altova Exif attributes are listed below.

#### Examples

- To access any one attribute, use the function like this:  
`image-exif-data(/MyImages/Image20141130.01)/@GPSLatitude`  
`image-exif-data(/MyImages/Image20141130.01)/@Geolocation`
- To access all the attributes, use the function like this:  
`image-exif-data(/MyImages/Image20141130.01)/@*`
- To access the names of all the attributes, use the following expression:  
`for $i in image-exif-data(/MyImages/Image20141130.01)/@* return name($i)`  
 This is useful to find out the names of the attributes returned by the function.

#### Altova Exif Attribute: Geolocation

The Altova XPath/XQuery Engine generates the custom attribute `Geolocation` from standard Exif metadata tags. `Geolocation` is a concatenation of four Exif tags: `GPSLatitude`, `GPSLatitudeRef`, `GPSLongitude`, `GPSLongitudeRef`, with units added (see *table below*).

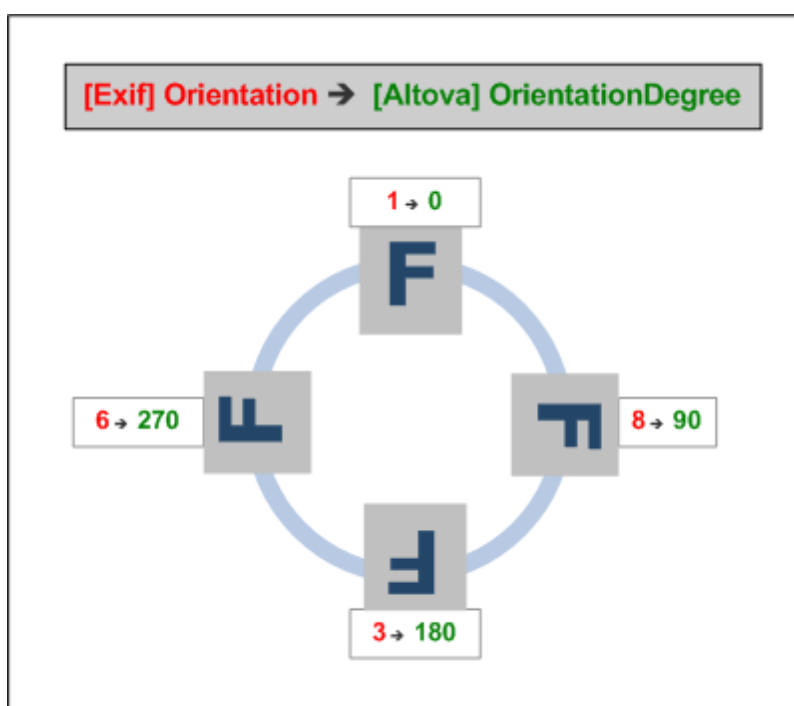
GPSLatitude	GPSLatitudeRef	GPSLongitude	GPSLongitudeRef	Geolocation
33 51 21.91	S	151 13 11.73	E	33°51'21.91"S 151°13'11.73"E



#### Altova Exif Attribute: *OrientationDegree*

The Altova XPath/XQuery Engine generates the custom attribute **OrientationDegree** from the Exif metadata tag **Orientation**.

**OrientationDegree** translates the standard Exif tag **Orientation** from an integer value (1, 8, 3, or 6) to the respective degree values of each (0, 90, 180, 270), as shown in the figure below. Note that there are no translations of the **Orientation** values of 2, 4, 5, 7. (These orientations are obtained by flipping image 1 across its vertical center axis to get the image with a value of 2, and then rotating this image in 90-degree jumps clockwise to get the values of 7, 4, and 5, respectively).



#### Listing of standard Exif meta tags

- ImageWidth
- ImageLength
- BitsPerSample
- Compression
- PhotometricInterpretation
- Orientation
- SamplesPerPixel
- PlanarConfiguration
- YCbCrSubSampling
- YCbCrPositioning
- XResolution
- YResolution
- ResolutionUnit
- StripOffsets



- RowsPerStrip
  - StripByteCounts
  - JPEGInterchangeFormat
  - JPEGInterchangeFormatLength
  - TransferFunction
  - WhitePoint
  - PrimaryChromaticities
  - YCbCrCoefficients
  - ReferenceBlackWhite
  - DateTime
  - ImageDescription
  - Make
  - Model
  - Software
  - Artist
  - Copyright
- 

- ExifVersion
- FlashpixVersion
- ColorSpace
- ComponentsConfiguration
- CompressedBitsPerPixel
- PixelXDimension
- PixelYDimension
- MakerNote
- UserComment
- RelatedSoundFile
- DateTimeOriginal
- DateTimeDigitized
- SubSecTime
- SubSecTimeOriginal
- SubSecTimeDigitized
- ExposureTime
- FNumber
- ExposureProgram
- SpectralSensitivity
- ISOSpeedRatings
- OECF
- ShutterSpeedValue
- ApertureValue
- BrightnessValue
- ExposureBiasValue
- MaxApertureValue
- SubjectDistance
- MeteringMode
- LightSource
- Flash
- FocalLength
- SubjectArea
- FlashEnergy
- SpatialFrequencyResponse
- FocalPlaneXResolution
- FocalPlaneYResolution
- FocalPlaneResolutionUnit







## XPath/XQuery Functions: Numeric

Altova's numeric extension functions can be used in XPath and XQuery expressions and provide additional functionality for the processing of data. The functions in this section can be used with Altova's **XPath 3.0** and **XQuery 3.0** engines. They are available in XPath/XQuery contexts.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, <http://www.altova.com/xslt-extensions>, and are indicated in this section with the prefix **altova:**, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

XPath functions (used in XPath expressions in XSLT):	XP1 XP2 XP3.1
XSLT functions (used in XPath expressions in XSLT):	XSLT1 XSLT2 XSLT3
XQuery functions (used in XQuery expressions in XQuery):	XQ1 XQ3.1

## Auto-numbering functions

▼ generate-auto-number [altova:]

**altova:generate-auto-number**(ID as xs:string, StartsWith as xs:double, Increment as xs:double, ResetOnChange as xs:string) as xs:integer XP1 XP2 XQ1 XP3.1 XQ3.1

Generates a number each time the function is called. The first number, which is generated the first time the function is called, is specified by the `StartsWith` argument. Each subsequent call to the function generates a new number, this number being incremented over the previously generated number by the value specified in the `Increment` argument. In effect, the `altova:generate-auto-number` function creates a counter having a name specified by the `ID` argument, with this counter being incremented each time the function is called. If the value of the `ResetOnChange` argument changes from that of the previous function call, then the value of the number to be generated is reset to the `StartsWith` value. Auto-numbering can also be reset by using the `altova:reset-auto-number` function.

### Examples

- **altova:generate-auto-number**("ChapterNumber", 1, 1, "SomeString") will return one number each time the function is called, starting with 1, and incrementing by 1 with each call to the function. As long as the fourth argument remains "SomeString" in each subsequent call, the incrementing will continue. When the value of the fourth argument changes, the counter (called ChapterNumber) will reset to 1. The value of ChapterNumber can also be reset by a call to the `altova:reset-auto-number` function, like this: `altova:reset-auto-number("ChapterNumber")`.



## ▼ reset-auto-number [altova:]

**altova:reset-auto-number**(**ID** as **xs:string**) **XP1 XP2 XQ1 XP3.1 XQ3.1**

This function resets the number of the auto-numbering counter named in the **ID** argument. The number is reset to the number specified by the **StartsWith** argument of the **altova:generate-auto-number** function that created the counter named in the **ID** argument.

▣ Examples

- **altova:reset-auto-number**("ChapterNumber") resets the number of the auto-numbering counter named ChapterNumber that was created by the **altova:generate-auto-number** function. The number is reset to the value of the **StartsWith** argument of the **altova:generate-auto-number** function that created ChapterNumber.

[\[ Top \]](#)

**Numeric functions**

## ▼ hex-string-to-integer [altova:]

**altova:hex-string-to-integer**(**HexString** as **xs:string**) as **xs:integer** **XP3.1 XQ3.1**

Takes a string argument that is the Base-16 equivalent of an integer in the decimal system (Base-10), and returns the decimal integer.

▣ Examples

- **altova:hex-string-to-integer**('1') returns 1
- **altova:hex-string-to-integer**('9') returns 9
- **altova:hex-string-to-integer**('A') returns 10
- **altova:hex-string-to-integer**('B') returns 11
- **altova:hex-string-to-integer**('F') returns 15
- **altova:hex-string-to-integer**('G') returns an error
- **altova:hex-string-to-integer**('10') returns 16
- **altova:hex-string-to-integer**('01') returns 1
- **altova:hex-string-to-integer**('20') returns 32
- **altova:hex-string-to-integer**('21') returns 33
- **altova:hex-string-to-integer**('5A') returns 90
- **altova:hex-string-to-integer**('USA') returns an error

## ▼ integer-to-hex-string [altova:]

**altova:integer-to-hex-string**(**Integer** as **xs:integer**) as **xs:string** **XP3.1 XQ3.1**

Takes an integer argument and returns its Base-16 equivalent as a string.

▣ Examples

- **altova:integer-to-hex-string**(1) returns '1'
- **altova:integer-to-hex-string**(9) returns '9'
- **altova:integer-to-hex-string**(10) returns 'A'
- **altova:integer-to-hex-string**(11) returns 'B'



- `altova:integer-to-hex-string(15)` returns 'F'
- `altova:integer-to-hex-string(16)` returns '10'
- `altova:integer-to-hex-string(32)` returns '20'
- `altova:integer-to-hex-string(33)` returns '21'
- `altova:integer-to-hex-string(90)` returns '5A'

[\[ Top \]](#)

## Number-formatting functions

### ▼ generate-auto-number [altova:]

```
altova:generate-auto-number(ID as xs:string, StartsWith as xs:double,
Increment as xs:double, ResetOnChange as xs:string) as xs:integer XP1 XP2 XQ1
XP3.1 XQ3.1
```

Generates a number each time the function is called. The first number, which is generated the first time the function is called, is specified by the `StartsWith` argument. Each subsequent call to the function generates a new number, this number being incremented over the previously generated number by the value specified in the `Increment` argument. In effect, the `altova:generate-auto-number` function creates a counter having a name specified by the `ID` argument, with this counter being incremented each time the function is called. If the value of the `ResetOnChange` argument changes from that of the previous function call, then the value of the number to be generated is reset to the `StartsWith` value. Auto-numbering can also be reset by using the `altova:reset-auto-number` function.

#### ▣ Examples

- `altova:generate-auto-number("ChapterNumber", 1, 1, "SomeString")` will return one number each time the function is called, starting with 1, and incrementing by 1 with each call to the function. As long as the fourth argument remains "SomeString" in each subsequent call, the incrementing will continue. When the value of the fourth argument changes, the counter (called `ChapterNumber`) will reset to 1. The value of `ChapterNumber` can also be reset by a call to the `altova:reset-auto-number` function, like this: `altova:reset-auto-number("ChapterNumber")`.

[\[ Top \]](#)

## *XPath/XQuery Functions: Sequence*

Altova's sequence extension functions can be used in XPath and XQuery expressions and provide additional functionality for the processing of data. The functions in this section can be used with Altova's **XPath 3.0** and **XQuery 3.0** engines. They are available in XPath/XQuery contexts.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, `http://www.altova.com/xslt-extensions`, and are indicated in this section



with the prefix **altova:**, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

<i>XPath functions (used in XPath expressions in XSLT):</i>	<b>XP1</b> <b>XP2</b> <b>XP3.1</b>
<i>XSLT functions (used in XPath expressions in XSLT):</i>	<b>XSLT1</b> <b>XSLT2</b> <b>XSLT3</b>
<i>XQuery functions (used in XQuery expressions in XQuery):</i>	<b>XQ1</b> <b>XQ3.1</b>

#### ▼ attributes [altova:]

**altova:attributes**(**AttributeName** as **xs:string**) as **attribute()**\* **XP3.1** **XQ3.1**

Returns all attributes that have a local name which is the same as the name supplied in the input argument, **AttributeName**. The search is case-sensitive and conducted along the **attribute::** axis. This means that the context node must be the parent element node.

##### ☐ Examples

- **altova:attributes**("MyAttribute") returns **MyAttribute()**\*

**altova:attributes**(**AttributeName** as **xs:string**, **SearchOptions** as **xs:string**) as **attribute()**\* **XP3.1** **XQ3.1**

Returns all attributes that have a local name which is the same as the name supplied in the input argument, **AttributeName**. The search is case-sensitive and conducted along the **attribute::** axis. The context node must be the parent element node. The second argument is a string containing option flags. Available flags are:

**r** = switches to a regular-expression search; **AttributeName** must then be a regular-expression search string;

**f** = If this option is specified, then **AttributeName** provides a full match; otherwise **AttributeName** need only partially match an attribute name to return that attribute. For example: if **f** is not specified, then **MyAtt** will return **MyAttribute**;

**i** = switches to a case-insensitive search;

**p** = includes the namespace prefix in the search; **AttributeName** should then contain the namespace prefix, for example: **altova:MyAttribute**.

The flags can be written in any order. Invalid flags will generate errors. One or more flags can be omitted. The empty string is allowed, and will produce the same effect as the function having only one argument (*previous signature*). However, an empty sequence is not allowed as the second argument.

##### ☐ Examples

- **altova:attributes**("MyAttribute", "rfip") returns **MyAttribute()**\*
- **altova:attributes**("MyAttribute", "pri") returns **MyAttribute()**\*
- **altova:attributes**("MyAtt", "rip") returns **MyAttribute()**\*
- **altova:attributes**("MyAttributes", "rfip") returns no match
- **altova:attributes**("MyAttribute", "") returns **MyAttribute()**\*
- **altova:attributes**("MyAttribute", "Rip") returns an unrecognized-flag error.
- **altova:attributes**("MyAttribute", ) returns a missing-second-argument error.

#### ▼ elements [altova:]



**altova:elements**(*ElementName* as *xs:string*) as *element()*\* XP3.1 XQ3.1

Returns all elements that have a local name which is the same as the name supplied in the input argument, *ElementName*. The search is case-sensitive and conducted along the *child::* axis. The context node must be the parent node of the element/s being searched for.

#### Examples

- **altova:elements**("MyElement") returns *MyElement()*\*

**altova:elements**(*ElementName* as *xs:string*, *SearchOptions* as *xs:string*) as *element()*\* XP3.1 XQ3.1

Returns all elements that have a local name which is the same as the name supplied in the input argument, *ElementName*. The search is case-sensitive and conducted along the *child::* axis. The context node must be the parent node of the element/s being searched for. The second argument is a string containing option flags. Available flags are:

- r** = switches to a regular-expression search; *ElementName* must then be a regular-expression search string;
  - f** = If this option is specified, then *ElementName* provides a full match; otherwise *ElementName* need only partially match an element name to return that element. For example: if **f** is not specified, then *MyElem* will return *MyElement*;
  - i** = switches to a case-insensitive search;
  - p** = includes the namespace prefix in the search; *ElementName* should then contain the namespace prefix, for example: *altova:MyElement*.
- The flags can be written in any order. Invalid flags will generate errors. One or more flags can be omitted. The empty string is allowed, and will produce the same effect as the function having only one argument (*previous signature*). However, an empty sequence is not allowed.

#### Examples

- **altova:elements**("MyElement", "rip") returns *MyElement()*\*
- **altova:elements**("MyElement", "pri") returns *MyElement()*\*
- **altova:elements**("MyElement", "") returns *MyElement()*\*
- **altova:attributes**("MyElem", "rip") returns *MyElement()*\*
- **altova:attributes**("MyElements", "rfip") returns no match
- **altova:elements**("MyElement", "Rip") returns an unrecognized-flag error.
- **altova:elements**("MyElement", ) returns a missing-second-argument error.

#### ▼ find-first [altova:]

**altova:find-first**((*Sequence* as *item()*\*), (*Condition* (*Sequence-Item* as *xs:boolean*))) as *item()*? XP3.1 XQ3.1

This function takes two arguments. The first argument is a sequence of one or more items of any datatype. The second argument, *Condition*, is a reference to an XPath function that takes one argument (has an arity of 1) and returns a boolean. Each item of *sequence* is submitted, in turn, to the function referenced in *Condition*. (*Remember*: This function takes a single argument.) The first *sequence* item that causes the function in *Condition* to evaluate to *true()* is returned as the result of **altova:find-first**, and the iteration stops.

#### Examples

- **altova:find-first**(5 to 10, function(\$a) {\$a mod 2 = 0}) returns  
*xs:integer* 6



The `condition` argument references the XPath 3.0 inline function, `function()`, which declares an inline function named `$a` and then defines it. Each item in the `sequence` argument of `altova:find-first` is passed, in turn, to `$a` as its input value. The input value is tested on the condition in the function definition (`$a mod 2 = 0`). The first input value to satisfy this condition is returned as the result of `altova:find-first` (in this case 6).

- `altova:find-first((1 to 10), (function($a) {$a+3=7}))` returns `xs:integer 4`

#### Further examples

If the file `C:\Temp\Customers.xml` exists:

- `altova:find-first( ("C:\Temp\Customers.xml", "http://www.altova.com/index.html"), (doc-available#1) )` returns `xs:string C:\Temp\Customers.xml`

If the file `C:\Temp\Customers.xml` does not exist, and `http://www.altova.com/index.html` exists:

- `altova:find-first( ("C:\Temp\Customers.xml", "http://www.altova.com/index.html"), (doc-available#1) )` returns `xs:string http://www.altova.com/index.html`

If the file `C:\Temp\Customers.xml` does not exist, and `http://www.altova.com/index.html` also does not exist:

- `altova:find-first( ("C:\Temp\Customers.xml", "http://www.altova.com/index.html"), (doc-available#1) )` returns no result

#### Notes about the examples given above

- The XPath 3.0 function, `doc-available`, takes a single string argument, which is used as a URI, and returns `true` if a document node is found at the submitted URI. (The document at the submitted URI must therefore be an XML document.)
- The `doc-available` function can be used for `condition`, the second argument of `altova:find-first`, because it takes only one argument (arity=1), because it takes an `item()` as input (a string which is used as a URI), and returns a boolean value.
- Notice that the `doc-available` function is only referenced, not called. The `#1` suffix that is attached to it indicates a function with an arity of 1. In its entirety `doc-available#1` simply means: *Use the `doc-available()` function that has arity=1, passing to it as its single argument, in turn, each of the items in the first sequence.* As a result, each of the two strings will be passed to `doc-available()`, which uses the string as a URI and tests whether a document node exists at the URI. If one does, the `doc-available()` evaluates to `true()` and that string is returned as the result of the `altova:find-first` function. *Note about the `doc-available()` function: Relative paths are resolved relative to the the current base URI, which is by default the URI of the XML document from which the function is loaded.*



▼ find-first-combination [altova:]

```
altova:find-first-combination((Seq-01 as item()*), (Seq-02 as item()*),
(Condition( Seq-01-Item, Seq-02-Item as xs:boolean)) as item()* XP3.1 XQ3.1
```

This function takes three arguments:

- The first two arguments, `seq-01` and `seq-02`, are sequences of one or more items of any datatype.
- The third argument, `condition`, is a reference to an XPath function that takes two arguments (has an arity of 2) and returns a boolean.

The items of `seq-01` and `seq-02` are passed in ordered pairs (one item from each sequence making up a pair) as the arguments of the function in `condition`. The pairs are ordered as follows.

```
If   Seq-01 = X1, X2, X3 ... Xn
And  Seq-02 = Y1, Y2, Y3 ... Yn
Then (X1 Y1), (X1 Y2), (X1 Y3) ... (X1 Yn), (X2 Y1), (X2 Y2) ... (Xn Yn)
```

The first ordered pair that causes the `condition` function to evaluate to `true()` is returned as the result of `altova:find-first-combination`. Note that: (i) If the `condition` function iterates through the submitted argument pairs and does not once evaluate to `true()`, then `altova:find-first-combination` returns *No results*; (ii) The result of `altova:find-first-combination` will always be a pair of items (of any datatype) or no item at all.

▣ Examples

- `altova:find-first-combination(11 to 20, 21 to 30, function($a, $b) {$a + $b = 32})` returns the sequence of `xs:integers` (11, 21)
- `altova:find-first-combination(11 to 20, 21 to 30, function($a, $b) {$a + $b = 33})` returns the sequence of `xs:integers` (11, 22)
- `altova:find-first-combination(11 to 20, 21 to 30, function($a, $b) {$a + $b = 34})` returns the sequence of `xs:integers` (11, 23)

▼ find-first-pair [altova:]

```
altova:find-first-pair((Seq-01 as item()*), (Seq-02 as item()*),
(Condition( Seq-01-Item, Seq-02-Item as xs:boolean)) as item()* XP3.1 XQ3.1
```

This function takes three arguments:

- The first two arguments, `seq-01` and `seq-02`, are sequences of one or more items of any datatype.
- The third argument, `condition`, is a reference to an XPath function that takes two arguments (has an arity of 2) and returns a boolean.

The items of `seq-01` and `seq-02` are passed in ordered pairs as the arguments of the function in `condition`. The pairs are ordered as follows.

```
If   Seq-01 = X1, X2, X3 ... Xn
And  Seq-02 = Y1, Y2, Y3 ... Yn
Then (X1 Y1), (X2 Y2), (X3 Y3) ... (Xn Yn)
```



The first ordered pair that causes the `Condition` function to evaluate to `true()` is returned as the result of `altova:find-first-pair`. Note that: (i) If the `Condition` function iterates through the submitted argument pairs and does not once evaluate to `true()`, then `altova:find-first-pair` returns *No results*; (ii) The result of `altova:find-first-pair` will always be a pair of items (of any datatype) or no item at all.

#### Examples

- `altova:find-first-pair(11 to 20, 21 to 30, function($a, $b) {$a+$b = 32})` returns the sequence of `xs:integers` (11, 21)
- `altova:find-first-pair(11 to 20, 21 to 30, function($a, $b) {$a+$b = 33})` returns *No results*

Notice from the two examples above that the ordering of the pairs is: (11, 21) (12, 22) (13, 23) ... (20, 30). This is why the second example returns *No results* (because no ordered pair gives a sum of 33).

#### ▼ find-first-pair-pos [altova:]

```
altova:find-first-pair-pos((Seq-01 as item()*), (Seq-02 as item()*),
(Condition( Seq-01-Item, Seq-02-Item as xs:boolean))) as xs:integer XP3.1 XQ3.1
```

This function takes three arguments:

- The first two arguments, `seq-01` and `seq-02`, are sequences of one or more items of any datatype.
- The third argument, `Condition`, is a reference to an XPath function that takes two arguments (has an arity of 2) and returns a boolean.

The items of `seq-01` and `seq-02` are passed in ordered pairs as the arguments of the function in `Condition`. The pairs are ordered as follows.

```
If   Seq-01 = X1, X2, X3 ... Xn
And  Seq-02 = Y1, Y2, Y3 ... Yn
Then (X1 Y1), (X2 Y2), (X3 Y3) ... (Xn Yn)
```

The index position of the first ordered pair that causes the `Condition` function to evaluate to `true()` is returned as the result of `altova:find-first-pair-pos`. Note that if the `Condition` function iterates through the submitted argument pairs and does not once evaluate to `true()`, then `altova:find-first-pair-pos` returns *No results*.

#### Examples

- `altova:find-first-pair-pos(11 to 20, 21 to 30, function($a, $b) {$a+$b = 32})` returns 1
- `altova:find-first-pair-pos(11 to 20, 21 to 30, function($a, $b) {$a+$b = 33})` returns *No results*

Notice from the two examples above that the ordering of the pairs is: (11, 21) (12, 22) (13, 23) ... (20, 30). In the first example, the first pair causes the `Condition` function to evaluate to `true()`, and so its index position in the sequence, 1, is returned. The second example returns *No results* because no pair gives a sum of 33.



## ▼ find-first-pos [altova:]

```
altova:find-first-pos((Sequence as item()*), (Condition( Sequence-Item as
xs:boolean))) as xs:integer XP3.1 XQ3.1
```

This function takes two arguments. The first argument is a sequence of one or more items of any datatype. The second argument, `Condition`, is a reference to an XPath function that takes one argument (has an arity of 1) and returns a boolean. Each item of `sequence` is submitted, in turn, to the function referenced in `Condition`. (*Remember:* This function takes a single argument.) The first `sequence` item that causes the function in `Condition` to evaluate to `true()` has its index position in `sequence` returned as the result of `altova:find-first-pos`, and the iteration stops.

☐ Examples

- `altova:find-first-pos(5 to 10, function($a) {$a mod 2 = 0})` returns  
`xs:integer 2`

The `Condition` argument references the XPath 3.0 inline function, `function()`, which declares an inline function named `$a` and then defines it. Each item in the `sequence` argument of `altova:find-first-pos` is passed, in turn, to `$a` as its input value. The input value is tested on the condition in the function definition (`$a mod 2 = 0`). The index position in the sequence of the first input value to satisfy this condition is returned as the result of `altova:find-first-pos` (in this case 2, since 6, the first value (in the sequence) to satisfy the condition, is at index position 2 in the sequence).

- `altova:find-first-pos((2 to 10), (function($a) {$a+3=7}))` returns  
`xs:integer 3`

Further examples

If the file `C:\Temp\Customers.xml` exists:

- `altova:find-first-pos( ("C:\Temp\Customers.xml", "http://www.altova.com/index.html"), (doc-available#1) )` returns `1`

If the file `C:\Temp\Customers.xml` does not exist, and `http://www.altova.com/index.html` exists:

- `altova:find-first-pos( ("C:\Temp\Customers.xml", "http://www.altova.com/index.html"), (doc-available#1) )` returns `2`

If the file `C:\Temp\Customers.xml` does not exist, and `http://www.altova.com/index.html` also does not exist:

- `altova:find-first-pos( ("C:\Temp\Customers.xml", "http://www.altova.com/index.html"), (doc-available#1) )` returns no result

Notes about the examples given above

- The XPath 3.0 function, `doc-available`, takes a single string argument, which is used as a URI, and returns `true` if a document node is found at the submitted URI. (The document at the submitted URI must therefore be an XML document.)



- The `doc-available` function can be used for `condition`, the second argument of `altova:find-first-pos`, because it takes only one argument (arity=1), because it takes an `item()` as input (a string which is used as a URI), and returns a boolean value.
- Notice that the `doc-available` function is only referenced, not called. The `#1` suffix that is attached to it indicates a function with an arity of 1. In its entirety `doc-available#1` simply means: *Use the `doc-available()` function that has arity=1, passing to it as its single argument, in turn, each of the items in the first sequence.* As a result, each of the two strings will be passed to `doc-available()`, which uses the string as a URI and tests whether a document node exists at the URI. If one does, the `doc-available()` function evaluates to `true()` and the index position of that string in the sequence is returned as the result of the `altova:find-first-pos` function. *Note about the `doc-available()` function: Relative paths are resolved relative to the the current base URI, which is by default the URI of the XML document from which the function is loaded.*

▼ `substitute-empty` [altova:]

```
altova:substitute-empty(FirstSequence as item()*, SecondSequence as item())
as item()* XP3.1 XQ3.1
```

If `FirstSequence` is empty, returns `SecondSequence`. If `FirstSequence` is not empty, returns `FirstSequence`.

▢ Examples

- `altova:substitute-empty( (1,2,3), (4,5,6) )` returns `(1,2,3)`
- `altova:substitute-empty( (), (4,5,6) )` returns `(4,5,6)`

## XPath/XQuery Functions: String

Altova's string extension functions can be used in XPath and XQuery expressions and provide additional functionality for the processing of data. The functions in this section can be used with Altova's **XPath 3.0** and **XQuery 3.0** engines. They are available in XPath/XQuery contexts.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, `http://www.altova.com/xslt-extensions`, and are indicated in this section with the prefix `altova:`, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

XPath functions (used in XPath expressions in XSLT):	XP1 XP2 XP3.1
------------------------------------------------------	---------------



XSLT functions (used in XPath expressions in XSLT):	XSLT1 XSLT2 XSLT3
XQuery functions (used in XQuery expressions in XQuery):	XQ1 XQ3.1

▼ camel-case [altova:]

**altova:camel-case**(**InputString** as **xs:string**) as **xs:string** XP3.1 XQ3.1

Returns the input string **InputString** in CamelCase. The string is analyzed using the regular expression `'\s'` (which is a shortcut for the whitespace character). The first non-whitespace character after a whitespace or sequence of consecutive whitespaces is capitalized. The first character in the output string is capitalized.

▣ Examples

- **altova:camel-case**("max") returns Max
- **altova:camel-case**("max max") returns Max Max
- **altova:camel-case**("file01.xml") returns File01.xml
- **altova:camel-case**("file01.xml file02.xml") returns File01.xml File02.xml
- **altova:camel-case**("file01.xml file02.xml") returns File01.xml File02.xml
- **altova:camel-case**("file01.xml -file02.xml") returns File01.xml -file02.xml

**altova:camel-case**(**InputString** as **xs:string**, **SplitChars** as **xs:string**, **IsRegex** as **xs:boolean**) as **xs:string** XP3.1 XQ3.1

Converts the input string **InputString** to camel case by using **splitChars** to determine the character/s that trigger the next capitalization. **splitChars** is used as a regular expression when **IsRegex** = **true()**, or as plain characters when **IsRegex** = **false()**. The first character in the output string is capitalized.

▣ Examples

- **altova:camel-case**("setname getname", "set|get", **true()**) returns setName getName
- **altova:camel-case**("altova\documents\testcases", "\", **false()**) returns Altova\Documents\Testcases

▼ char [altova:]

**altova:char**(**Position** as **xs:integer**) as **xs:string** XP3.1 XQ3.1

Returns a string containing the character at the position specified by the **Position** argument, in the string obtained by converting the value of the context item to **xs:string**. The result string will be empty if no character exists at the index submitted by the **Position** argument.

▣ Examples

If the context item is 1234ABCD:

- **altova:char**(2) returns 2
- **altova:char**(5) returns A
- **altova:char**(9) returns the empty string.
- **altova:char**(-2) returns the empty string.



```
altova:char(InputString as xs:string, Position as xs:integer) as xs:string
XP3.1 XQ3.1
```

Returns a string containing the character at the position specified by the `Position` argument, in the string submitted as the `InputString` argument. The result string will be empty if no character exists at the index submitted by the `Position` argument.

▣ Examples

- `altova:char("2014-01-15", 5)` returns -
- `altova:char("USA", 1)` returns U
- `altova:char("USA", 10)` returns the empty string.
- `altova:char("USA", -2)` returns the empty string.

▼ first-chars [altova:]

```
altova:first-chars(X-Number as xs:integer) as xs:string XP3.1 XQ3.1
```

Returns a string containing the first `X-Number` of characters of the string obtained by converting the value of the context item to `xs:string`.

▣ Examples

If the context item is 1234ABCD:

- `altova:first-chars(2)` returns 12
- `altova:first-chars(5)` returns 1234A
- `altova:first-chars(9)` returns 1234ABCD

```
altova:first-chars(InputString as xs:string, X-Number as xs:integer) as
xs:string XP3.1 XQ3.1
```

Returns a string containing the first `X-Number` of characters of the string submitted as the `InputString` argument.

▣ Examples

- `altova:first-chars("2014-01-15", 5)` returns 2014-
- `altova:first-chars("USA", 1)` returns U

▼ last-chars [altova:]

```
altova:last-chars(X-Number as xs:integer) as xs:string XP3.1 XQ3.1
```

Returns a string containing the last `X-Number` of characters of the string obtained by converting the value of the context item to `xs:string`.

▣ Examples

If the context item is 1234ABCD:

- `altova:last-chars(2)` returns CD
- `altova:last-chars(5)` returns 4ABCD
- `altova:last-chars(9)` returns 1234ABCD

```
altova:last-chars(InputString as xs:string, X-Number as xs:integer) as
xs:string XP3.1 XQ3.1
```

Returns a string containing the last `X-Number` of characters of the string submitted as the `InputString` argument.



#### ▣ Examples

- `altova:last-chars("2014-01-15", 5)` returns `01-15`
- `altova:last-chars("USA", 10)` returns `USA`

#### ▼ `pad-string-left` [altova:]

```
altova:pad-string-left(StringToPad as xs:string, StringLength as xs:integer,
PadCharacter as xs:string) as xs:string XP3.1 XQ3.1
```

The `PadCharacter` argument is a single character. It is padded to the left of the string to increase the number of characters in `StringToPad` so that this number equals the integer value of the `StringLength` argument. The `StringLength` argument can have any integer value (positive or negative), but padding will occur only if the value of `StringLength` is greater than the number of characters in `StringToPad`. If `StringToPad` has more characters than the value of `StringLength`, then `StringToPad` is left unchanged.

#### ▣ Examples

- `altova:pad-string-left('AP', 1, 'Z')` returns `'AP'`
- `altova:pad-string-left('AP', 2, 'Z')` returns `'AP'`
- `altova:pad-string-left('AP', 3, 'Z')` returns `'ZAP'`
- `altova:pad-string-left('AP', 4, 'Z')` returns `'ZZAP'`
- `altova:pad-string-left('AP', -3, 'Z')` returns `'AP'`
- `altova:pad-string-left('AP', 3, 'YZ')` returns a `pad-character-too-long` error

#### ▼ `pad-string-right` [altova:]

```
altova:pad-string-right(StringToPad as xs:string, StringLength as
xs:integer, PadCharacter as xs:string) as xs:string XP3.1 XQ3.1
```

The `PadCharacter` argument is a single character. It is padded to the right of the string to increase the number of characters in `StringToPad` so that this number equals the integer value of the `StringLength` argument. The `StringLength` argument can have any integer value (positive or negative), but padding will occur only if the value of `StringLength` is greater than the number of characters in `StringToPad`. If `StringToPad` has more characters than the value of `StringLength`, then `StringToPad` is left unchanged.

#### ▣ Examples

- `altova:pad-string-right('AP', 1, 'Z')` returns `'AP'`
- `altova:pad-string-right('AP', 2, 'Z')` returns `'AP'`
- `altova:pad-string-right('AP', 3, 'Z')` returns `'APZ'`
- `altova:pad-string-right('AP', 4, 'Z')` returns `'APZZ'`
- `altova:pad-string-right('AP', -3, 'Z')` returns `'AP'`
- `altova:pad-string-right('AP', 3, 'YZ')` returns a `pad-character-too-long` error

#### ▼ `repeat-string` [altova:]

```
altova:repeat-string(InputString as xs:string, Repeats as xs:integer) as
xs:string XP2 XQ1 XP3.1 XQ3.1
```

Generates a string that is composed of the first `InputString` argument repeated `Repeats` number of times.

#### ▣ Examples



- `altova:repeat-string("Altova #", 3)` returns `"Altova #Altova #Altova #"`

#### ▼ substring-after-last [altova:]

`altova:substring-after-last(MainString as xs:string, CheckString as xs:string) as xs:string` **XP3.1 XQ3.1**

If CheckString is found in MainString, then the substring that occurs after CheckString in MainString is returned. If CheckString is not found in MainString, then the empty string is returned. If CheckString is an empty string, then MainString is returned in its entirety. If there is more than one occurrence of CheckString in MainString, then the substring after the last occurrence of CheckString is returned.

##### ▢ Examples

- `altova:substring-after-last('ABCDEFGH', 'B')` returns `'CDEFGH'`
- `altova:substring-after-last('ABCDEFGH', 'BC')` returns `'DEFGH'`
- `altova:substring-after-last('ABCDEFGH', 'BD')` returns `''`
- `altova:substring-after-last('ABCDEFGH', 'Z')` returns `''`
- `altova:substring-after-last('ABCDEFGH', '')` returns `'ABCDEFGH'`
- `altova:substring-after-last('ABCD-ABCD', 'B')` returns `'CD'`
- `altova:substring-after-last('ABCD-ABCD-ABCD', 'BCD')` returns `''`

#### ▼ substring-before-last [altova:]

`altova:substring-before-last(MainString as xs:string, CheckString as xs:string) as xs:string` **XP3.1 XQ3.1**

If CheckString is found in MainString, then the substring that occurs before CheckString in MainString is returned. If CheckString is not found in MainString, or if CheckString is an empty string, then the empty string is returned. If there is more than one occurrence of CheckString in MainString, then the substring before the last occurrence of CheckString is returned.

##### ▢ Examples

- `altova:substring-before-last('ABCDEFGH', 'B')` returns `'A'`
- `altova:substring-before-last('ABCDEFGH', 'BC')` returns `'A'`
- `altova:substring-before-last('ABCDEFGH', 'BD')` returns `''`
- `altova:substring-before-last('ABCDEFGH', 'Z')` returns `''`
- `altova:substring-before-last('ABCDEFGH', '')` returns `''`
- `altova:substring-before-last('ABCD-ABCD', 'B')` returns `'ABCD-A'`
- `altova:substring-before-last('ABCD-ABCD-ABCD', 'ABCD')` returns `'ABCD-ABCD-'`

#### ▼ substring-pos [altova:]

`altova:substring-pos(StringToCheck as xs:string, StringToFind as xs:string) as xs:integer` **XP3.1 XQ3.1**

Returns the character position of the first occurrence of StringToFind in the string StringToCheck. The character position is returned as an integer. The first character of StringToCheck has the position 1. If StringToFind does not occur within StringToCheck, the integer 0 is returned. To check for the second or a later occurrence of StringToCheck,



use the next signature of this function.

▣ Examples

- `altova:substring-pos('Altova', 'to')` returns 3
- `altova:substring-pos('Altova', 'tov')` returns 3
- `altova:substring-pos('Altova', 'tv')` returns 0
- `altova:substring-pos('AltovaAltova', 'to')` returns 3

`altova:substring-pos(StringToCheck as xs:string, StringToFind as xs:string, Integer as xs:integer) as xs:integer XP3.1 XQ3.1`

Returns the character position of StringToFind in the string, StringToCheck. The search for StringToFind starts from the character position given by the Integer argument; the character substring before this position is not searched. The returned integer, however, is the position of the found string within the *entire* string, StringToCheck. This signature is useful for finding the second or a later position of a string that occurs multiple times with the StringToCheck. If StringToFind does not occur within StringToCheck, the integer 0 is returned.

▣ Examples

- `altova:substring-pos('Altova', 'to', 1)` returns 3
- `altova:substring-pos('Altova', 'to', 3)` returns 3
- `altova:substring-pos('Altova', 'to', 4)` returns 0
- `altova:substring-pos('Altova-Altova', 'to', 0)` returns 3
- `altova:substring-pos('Altova-Altova', 'to', 4)` returns 10

▼ trim-string [altova:]

`altova:trim-string(InputString as xs:string) as xs:string XP3.1 XQ3.1`

This function takes an xs:string argument, removes any leading and trailing whitespace, and returns a "trimmed" xs:string.

▣ Examples

- `altova:trim-string(" Hello World ")` returns "Hello World"
- `altova:trim-string("Hello World ")` returns "Hello World"
- `altova:trim-string(" Hello World")` returns "Hello World"
- `altova:trim-string("Hello World")` returns "Hello World"
- `altova:trim-string("Hello World")` returns "Hello World"

▼ trim-string-left [altova:]

`altova:trim-string-left(InputString as xs:string) as xs:string XP3.1 XQ3.1`

This function takes an xs:string argument, removes any leading whitespace, and returns a left-trimmed xs:string.

▣ Examples

- `altova:trim-string-left(" Hello World ")` returns "Hello World "
- `altova:trim-string-left("Hello World ")` returns "Hello World "
- `altova:trim-string-left(" Hello World")` returns "Hello World"
- `altova:trim-string-left("Hello World")` returns "Hello World"
- `altova:trim-string-left("Hello World")` returns "Hello World"



## ▼ trim-string-right [altova:]

**altova:trim-string-right**(**InputString** as **xs:string**) as **xs:string** **XP3.1** **XQ3.1**

This function takes an **xs:string** argument, removes any trailing whitespace, and returns a right-trimmed **xs:string**.

▣ Examples

- **altova:trim-string-right**(" Hello World ") returns " Hello World"
- **altova:trim-string-right**("Hello World ") returns "Hello World"
- **altova:trim-string-right**(" Hello World") returns " Hello World"
- **altova:trim-string-right**("Hello World") returns "Hello World"
- **altova:trim-string-right**("Hello World") returns "Hello World"

### *XPath/XQuery Functions: Miscellaneous*

The following general purpose XPath/XQuery extension functions are supported in the current version of MapForce and can be used in (i) XPath expressions in an XSLT context, or (ii) XQuery expressions in an XQuery document.

Note about naming of functions and language applicability

Altova extension functions can be used in XPath/XQuery expressions. They provide additional functionality to the functionality that is available in the standard library of XPath, XQuery, and XSLT functions. Altova extension functions are in the **Altova extension functions namespace**, **<http://www.altova.com/xslt-extensions>**, and are indicated in this section with the prefix **altova:**, which is assumed to be bound to this namespace. Note that, in future versions of your product, support for a function might be discontinued or the behavior of individual functions might change. Consult the documentation of future releases for information about support for Altova extension functions in that release.

<i>XPath functions (used in XPath expressions in XSLT):</i>	<b>XP1</b> <b>XP2</b> <b>XP3.1</b>
<i>XSLT functions (used in XPath expressions in XSLT):</i>	<b>XSLT1</b> <b>XSLT2</b> <b>XSLT3</b>
<i>XQuery functions (used in XQuery expressions in XQuery):</i>	<b>XQ1</b> <b>XQ3.1</b>

## ▼ get-temp-folder [altova:]

**altova:get-temp-folder**() as **xs:string** **XP2** **XQ1** **XP3.1** **XQ3.1**

This function takes no argument. It returns the path to the temporary folder of the current user.

▣ Examples

- **altova:get-temp-folder**() would return, on a Windows machine, something like **C:\Users\<UserName>\AppData\Local\Temp\** as an **xs:string**.

## ▼ generate-guid [altova:]



```
altova:generate-guid() as xs:string XP2 XQ1 XP3.1 XQ3.1
```

Generates a unique string GUID string.

▣ Examples

- `altova:generate-guid()` returns (for example) `85F971DA-17F3-4E4E-994E-99137873ACCD`

[\[ Top \]](#)

### 11.1.2.2 Miscellaneous Extension Functions

There are several ready-made functions in programming languages such as Java and C# that are not available as XQuery/XPath functions or as XSLT functions. A good example would be the math functions available in Java, such as `sin()` and `cos()`. If these functions were available to the designers of XSLT stylesheets and XQuery queries, it would increase the application area of stylesheets and queries and greatly simplify the tasks of stylesheet creators. The XSLT and XQuery engines used in a number of Altova products support the use of extension functions in [Java](#) and [.NET](#), as well as [MSXSL scripts for XSLT](#). This section describes how to use extension functions and MSXSL scripts in your XSLT stylesheets and XQuery documents. The available extension functions are organized into the following sections:

- [Java Extension Functions](#)
- [.NET Extension Functions](#)
- [MSXSL Scripts for XSLT](#)

The two main issues considered in the descriptions are: (i) how functions in the respective libraries are called; and (ii) what rules are followed for converting arguments in a function call to the required input format of the function, and what rules are followed for the return conversion (function result to XSLT/XQuery data object).

#### Requirements

For extension functions support, a Java Runtime Environment (for access to Java functions) and .NET Framework 2.0 (minimum, for access to .NET functions) must be installed on the machine running the XSLT transformation or XQuery execution, or must be accessible for the transformations.

#### Java Extension Functions

A Java extension function can be used within an XPath or XQuery expression to invoke a Java constructor or call a Java method (static or instance).

A field in a Java class is considered to be a method without any argument. A field can be static or instance. How to access fields is described in the respective sub-sections, static and instance.

This section is organized into the following sub-sections:



- [Java: Constructors](#)
- [Java: Static Methods and Static Fields](#)
- [Java: Instance Methods and Instance Fields](#)
- [Datatypes: XPath/XQuery to Java](#)
- [Datatypes: Java to XPath/XQuery](#)

### Form of the extension function

The extension function in the XPath/XQuery expression must have the form `prefix:fname()`.

- The `prefix:` part identifies the extension function as a Java function. It does so by associating the extension function with an in-scope namespace declaration, the URI of which must begin with `java:` (*see below for examples*). The namespace declaration should identify a Java class, for example: `xmlns:myns="java:java.lang.Math"`. However, it could also simply be: `xmlns:myns="java"` (without a colon), with the identification of the Java class being left to the `fname()` part of the extension function.
- The `fname()` part identifies the Java method being called, and supplies the arguments for the method (*see below for examples*). However, if the namespace URI identified by the `prefix:` part does not identify a Java class (*see preceding point*), then the Java class should be identified in the `fname()` part, before the class and separated from the class by a period (*see the second XSLT example below*).

**Note:** The class being called must be on the classpath of the machine.

### XSLT example

Here are two examples of how a static method can be called. In the first example, the class name (`java.lang.Math`) is included in the namespace URI and, therefore, must not be in the `fname()` part. In the second example, the `prefix:` part supplies the prefix `java:` while the `fname()` part identifies the class as well as the method.

```
<xsl:value-of xmlns:jMath="java:java.lang.Math"
  select="jMath:cos(3.14)" />

<xsl:value-of xmlns:jmath="java"
  select="jmath:java.lang.Math.cos(3.14)" />
```

The method named in the extension function (`cos()` in the example above) must match the name of a public static method in the named Java class (`java.lang.Math` in the example above).

### XQuery example

Here is an XQuery example similar to the XSLT example above:

```
<cosine xmlns:jMath="java:java.lang.Math">
  { jMath:cos(3.14) }
</cosine>
```



## User-defined Java classes

If you have created your own Java classes, methods in these classes are called differently according to: (i) whether the classes are accessed via a JAR file or a class file, and (ii) whether these files (JAR or class) are located in the current directory (the same directory as the XSLT or XQuery document) or not. How to locate these files is described in the sections [User-Defined Class Files](#) and [User-Defined Jar Files](#). Note that paths to class files not in the current directory and to all JAR files must be specified.

### User-Defined Class Files

If access is via a class file, then there are two possibilities:

- The class file is in a package. The XSLT or XQuery file is in the same folder as the Java package. ([See example below.](#))
- The class file is not packaged. The XSLT or XQuery file is in the same folder as the class file. ([See example below.](#))
- The class file is in a package. The XSLT or XQuery file is at some random location. ([See example below.](#))
- The class file is not packaged. The XSLT or XQuery file is at some random location. ([See example below.](#))

Consider the case where the class file is not packaged and is in the same folder as the XSLT or XQuery document. In this case, since all classes in the folder are found, the file location does not need to be specified. The syntax to identify a class is:

```
java:classname
```

where

java: indicates that a user-defined Java function is being called; (Java classes in the current directory will be loaded by default)

classname is the name of the required method's class

The class is identified in a namespace URI, and the namespace is used to prefix a method call.

### Class file packaged, XSLT/XQuery file in same folder as Java package

The example below calls the `getVehicleType()` method of the `Car` class of the `com.altova.extfunc` package. The `com.altova.extfunc` package is in the folder `JavaProject`. The XSLT file is also in the folder `JavaProject`.

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:car="java:com.altova.extfunc.Car" >
```



```

<xsl:output exclude-result-prefixes="fn car xsl fo xs"/>

<xsl:template match="/">
  <a>
    <xsl:value-of select="car:getVehicleType()" />
  </a>
</xsl:template>

</xsl:stylesheet>

```

### Class file not packaged, XSLT/XQuery file in same folder as class file

The example below calls the `getVehicleType()` method of the `Car` class of the `com.altova.extfunc` package. The `Car` class file is in the following folder location: `JavaProject/com/altova/extfunc`. The XSLT file is also in the folder `JavaProject/com/altova/extfunc`.

```

<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:car="java:Car" >
<xsl:output exclude-result-prefixes="fn car xsl fo xs"/>

<xsl:template match="/">
  <a>
    <xsl:value-of select="car:getVehicleType()" />
  </a>
</xsl:template>

</xsl:stylesheet>

```

### Class file packaged, XSLT/XQuery file at any location

The example below calls the `getCarColor()` method of the `Car` class of the `com.altova.extfunc` package. The `com.altova.extfunc` package is in the folder `JavaProject`. The XSLT file is at any location. In this case, the location of the package must be specified within the URI as a query string. The syntax is:

```
java:classname[?path=uri-of-package]
```

*where*

`java:` indicates that a user-defined Java function is being called  
`uri-of-package` is the URI of the Java package  
`classname` is the name of the required method's class

The class is identified in a namespace URI, and the namespace is used to prefix a method call. The example below shows how to access a class file that is located in another directory than the current directory.



```

<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:car="java:com.altova.extfunc.Car?path=file:///C:/
JavaProject/" >

  <xsl:output exclude-result-prefixes="fn car xsl xs"/>

  <xsl:template match="/">
    <xsl:variable name="myCar" select="car:new('red')"/>
    <a><xsl:value-of select="car:getCarColor($myCar)"/></a>
  </xsl:template>

</xsl:stylesheet>

```

### Class file not packaged, XSLT/XQuery file at any location

The example below calls the `getCarColor()` method of the `Car` class of the `com.altova.extfunc` package. The `com.altova.extfunc` package is in the folder `JavaProject`. The XSLT file is at any location. The location of the class file is specified within the namespace URI as a query string.

The syntax is:

```
java:classname[?path=uri-of-classfile]
```

where

`java:` indicates that a user-defined Java function is being called  
`uri-of-classfile` is the URI of the folder containing the class file  
`classname` is the name of the required method's class

The class is identified in a namespace URI, and the namespace is used to prefix a method call. The example below shows how to access a class file that is located in another directory than the current directory.

```

<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:car="java:Car?path=file:///C:/JavaProject/com/altova/
extfunc/" >

  <xsl:output exclude-result-prefixes="fn car xsl xs"/>

  <xsl:template match="/">
    <xsl:variable name="myCar" select="car:new('red')"/>
    <a><xsl:value-of select="car:getCarColor($myCar)"/></a>
  </xsl:template>

</xsl:stylesheet>

```

**Note:** When a path is supplied via the extension function, the path is added to the `ClassLoader`.



## User-Defined Jar Files

If access is via a JAR file, the URI of the JAR file must be specified using the following syntax:

```
xmlns:classNS="java:classname?path=jar:uri-of-jarfile!/"
```

The method is then called by using the prefix of the namespace URI that identifies the class: `classNS:method()`

*In the above:*

java: indicates that a Java function is being called  
 classname is the name of the user-defined class  
 ? is the separator between the classname and the path  
 path=jar: indicates that a path to a JAR file is being given  
 uri-of-jarfile is the URI of the jar file  
 !/ is the end delimiter of the path  
 classNS:method() is the call to the method

Alternatively, the classname can be given with the method call. Here are two examples of the syntax:

```
xmlns:ns1="java:docx.layout.pages?path=jar:file:///c:/projects/
docs/docx.jar!/"
ns1:main()

xmlns:ns2="java?path=jar:file:///c:/projects/docs/docx.jar!/"
ns2:docx.layout.pages.main()
```

Here is a complete XSLT example that uses a JAR file to call a Java extension function:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:car="java?path=jar:file:///C:/test/Car1.jar!/" >
  <xsl:output exclude-result-prefixes="fn car xsl xs"/>

  <xsl:template match="/">
    <xsl:variable name="myCar" select="car:Car1.new('red')"/>
    <a><xsl:value-of select="car:Car1.getCarColor($myCar)"/></a>
  </xsl:template>

  <xsl:template match="car"/>

</xsl:stylesheet>
```

**Note:** When a path is supplied via the extension function, the path is added to the ClassLoader.



## Java: Constructors

An extension function can be used to call a Java constructor. All constructors are called with the pseudo-function `new()`.

If the result of a Java constructor call can be [implicitly converted to XPath/XQuery datatypes](#), then the Java extension function will return a sequence that is an XPath/XQuery datatype. If the result of a Java constructor call cannot be converted to a suitable XPath/XQuery datatype, then the constructor creates a wrapped Java object with a type that is the name of the class returning that Java object. For example, if a constructor for the class `java.util.Date` is called (`java.util.Date.new()`), then an object having a type `java.util.Date` is returned. The lexical format of the returned object may not match the lexical format of an XPath datatype and the value would therefore need to be converted to the lexical format of the required XPath datatype and then to the required XPath datatype.

There are two things that can be done with a Java object created by a constructor:

- It can be assigned to a variable:  

```
<xsl:variable name="currentdate" select="date:new()"
xmlns:date="java:java.util.Date" />
```
- It can be passed to an extension function (see [Instance Method and Instance Fields](#)):  

```
<xsl:value-of select="date:toString(date:new())"
xmlns:date="java:java.util.Date" />
```

## Java: Static Methods and Static Fields

A static method is called directly by its Java name and by supplying the arguments for the method. Static fields (methods that take no arguments), such as the constant-value fields `E` and `PI`, are accessed without specifying any argument.

## XSLT examples

Here are some examples of how static methods and fields can be called:

```
<xsl:value-of xmlns:jMath="java:java.lang.Math"
select="jMath:cos(3.14)" />

<xsl:value-of xmlns:jMath="java:java.lang.Math"
select="jMath:cos(jMath:PI())" />

<xsl:value-of xmlns:jMath="java:java.lang.Math"
select="jMath:E() * jMath:cos(3.14)" />
```

Notice that the extension functions above have the form `prefix:fname()`. The prefix in all three cases is `jMath:`, which is associated with the namespace URI `java:java.lang.Math`. (The namespace URI must begin with `java:`. In the examples above it is extended to contain the class name (`java.lang.Math`.) The `fname()` part of the extension functions must match the name of a public class (e.g. `java.lang.Math`) followed by the name of a public static method with its



argument/s (such as `cos(3.14)`) or a public static field (such as `PI()`).

In the examples above, the class name has been included in the namespace URI. If it were not contained in the namespace URI, then it would have to be included in the `fname()` part of the extension function. For example:

```
<xsl:value-of xmlns:java="java:"
              select="java:java.lang.Math.cos(3.14)" />
```

### XQuery example

A similar example in XQuery would be:

```
<cosine xmlns:jMath="java:java.lang.Math">
  {jMath:cos(3.14)}
</cosine>
```

### Java: Instance Methods and Instance Fields

An instance method has a Java object passed to it as the first argument of the method call. Such a Java object typically would be created by using an extension function (for example a constructor call) or a stylesheet parameter/variable. An XSLT example of this kind would be:

```
<xsl:stylesheet version="1.0" exclude-result-prefixes="date"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:date="java:java.util.Date"
  xmlns:jl="java:java.lang">
  <xsl:param name="CurrentDate" select="date:new()" />
  <xsl:template match="/">
    <enrollment institution-id="Altova School"
      date="{date:toString($CurrentDate)}"
      type="
{jl:Object.toString(jl:Object.getClass( date:new() ))}">
    </enrollment>
  </xsl:template>
</xsl:stylesheet>
```

In the example above, the value of the node `enrollment/@type` is created as follows:

1. An object is created with a constructor for the class `java.util.Date` (with the `date:new()` constructor).
2. This Java object is passed as the argument of the `jl.Object.getClass` method.
3. The object obtained by the `getClass` method is passed as the argument to the `jl.Object.toString` method.

The result (the value of `@type`) will be a string having the value: `java.util.Date`.

An instance field is theoretically different from an instance method in that it is not a Java object per se that is passed as an argument to the instance field. Instead, a parameter or variable is passed as the argument. However, the parameter/variable may itself contain the value returned by a Java object. For example, the parameter `CurrentDate` takes the value returned by a constructor



for the class `java.util.Date`. This value is then passed as an argument to the instance method `date:toString` in order to supply the value of `/enrollment/@date`.

### Datatypes: XPath/XQuery to Java

When a Java function is called from within an XPath/XQuery expression, the datatype of the function's arguments is important in determining which of multiple Java classes having the same name is called.

In Java, the following rules are followed:

- If there is more than one Java method with the same name, but each has a different number of arguments than the other/s, then the Java method that best matches the number of arguments in the function call is selected.
- The XPath/XQuery string, number, and boolean datatypes (see list below) are implicitly converted to a corresponding Java datatype. If the supplied XPath/XQuery type can be converted to more than one Java type (for example, `xs:integer`), then that Java type is selected which is declared for the selected method. For example, if the Java method being called is `fx(decimal)` and the supplied XPath/XQuery datatype is `xs:integer`, then `xs:integer` will be converted to Java's `decimal` datatype.

The table below lists the implicit conversions of XPath/XQuery string, number, and boolean types to Java datatypes.

<code>xs:string</code>	<code>java.lang.String</code>
<code>xs:boolean</code>	<code>boolean (primitive)</code> , <code>java.lang.Boolean</code>
<code>xs:integer</code>	<code>int</code> , <code>long</code> , <code>short</code> , <code>byte</code> , <code>float</code> , <code>double</code> , and the wrapper classes of these, such as <code>java.lang.Integer</code>
<code>xs:float</code>	<code>float (primitive)</code> , <code>java.lang.Float</code> , <code>double (primitive)</code>
<code>xs:double</code>	<code>double (primitive)</code> , <code>java.lang.Double</code>
<code>xs:decimal</code>	<code>float (primitive)</code> , <code>java.lang.Float</code> , <code>double(primitive)</code> , <code>java.lang.Double</code>

Subtypes of the XML Schema datatypes listed above (and which are used in XPath and XQuery) will also be converted to the Java type/s corresponding to that subtype's ancestor type.

In some cases, it might not be possible to select the correct Java method based on the supplied information. For example, consider the following case.

- The supplied argument is an `xs:untypedAtomic` value of 10 and it is intended for the method `myMethod(float)`.
- However, there is another method in the class which takes an argument of another datatype: `myMethod(double)`.
- Since the method names are the same and the supplied type (`xs:untypedAtomic`) could be converted correctly to either `float` or `double`, it is possible that `xs:untypedAtomic` is



- converted to `double` instead of `float`.
- Consequently the method selected will not be the required method and might not produce the expected result. To work around this, you can create a user-defined method with a different name and use this method.

Types that are not covered in the list above (for example `xs:date`) will not be converted and will generate an error. However, note that in some cases, it might be possible to create the required Java type by using a Java constructor.

### Datatypes: Java to XPath/XQuery

When a Java method returns a value, the datatype of the value is a string, numeric or boolean type, then it is converted to the corresponding XPath/XQuery type. For example, Java's `java.lang.Boolean` and `boolean` datatypes are converted to `xs:boolean`.

One-dimensional arrays returned by functions are expanded to a sequence. Multi-dimensional arrays will not be converted, and should therefore be wrapped.

When a wrapped Java object or a datatype other than string, numeric or boolean is returned, you can ensure conversion to the required XPath/XQuery type by first using a Java method (e.g. `toString`) to convert the Java object to a string. In XPath/XQuery, the string can be modified to fit the lexical representation of the required type and then converted to the required type (for example, by using the `cast as` expression).

### *.NET Extension Functions*

If you are working on the .NET platform on a Windows machine, you can use extension functions written in any of the .NET languages (for example, C#). A .NET extension function can be used within an XPath or XQuery expression to invoke a constructor, property, or method (static or instance) within a .NET class.

A property of a .NET class is called using the syntax `get_PropertyName()`.

This section is organized into the following sub-sections:

- [.NET: Constructors](#)
- [.NET: Static Methods and Static Fields](#)
- [.NET: Instance Methods and Instance Fields](#)
- [Datatypes: XPath/XQuery to .NET](#)
- [Datatypes: .NET to XPath/XQuery](#)

### Form of the extension function

The extension function in the XPath/XQuery expression must have the form `prefix:fname()`.

- The `prefix:` part is associated with a URI that identifies the .NET class being addressed.
- The `fname()` part identifies the constructor, property, or method (static or instance) within



- the .NET class, and supplies any argument/s, if required.
- The URI must begin with `clitype:` (which identifies the function as being a .NET extension function).
- The `prefix:fname()` form of the extension function can be used with system classes and with classes in a loaded assembly. However, if a class needs to be loaded, additional parameters containing the required information will have to be supplied.

## Parameters

To load an assembly, the following parameters are used:

<code>asm</code>	The name of the assembly to be loaded.
<code>ver</code>	The version number (maximum of four integers separated by periods).
<code>sn</code>	The key token of the assembly's strong name (16 hex digits).
<code>from</code>	A URI that gives the location of the assembly (DLL) to be loaded. If the URI is relative, it is relative to the XSLT or XQuery document. If this parameter is present, any other parameter is ignored.
<code>partialname</code>	The partial name of the assembly. It is supplied to <code>Assembly.LoadWith.PartialName()</code> , which will attempt to load the assembly. If <code>partialname</code> is present, any other parameter is ignored.
<code>loc</code>	The locale, for example, <code>en-US</code> . The default is <code>neutral</code> .

If the assembly is to be loaded from a DLL, use the `from` parameter and omit the `sn` parameter. If the assembly is to be loaded from the Global Assembly Cache (GAC), use the `sn` parameter and omit the `from` parameter.

A question mark must be inserted before the first parameter, and parameters must be separated by a semi-colon. The parameter name gives its value with an equals sign (*see example below*).

## Examples of namespace declarations

An example of a namespace declaration in XSLT that identifies the system class `System.Environment`:

```
xmlns:myns="clitype:System.Environment"
```

An example of a namespace declaration in XSLT that identifies the class to be loaded as `Trade.Forward.Scrip`:

```
xmlns:myns="clitype:Trade.Forward.Scrip?asm=forward;version=10.6.2.1"
```

An example of a namespace declaration in XQuery that identifies the system class `MyManagedDLL.testClass`:. Two cases are distinguished:

- When the assembly is loaded from the GAC:

```
declare namespace cs="clitype:MyManagedDLL.testClass?asm=MyManagedDLL;
ver=1.2.3.4;loc=neutral;sn=b9f091b72dccbfa8";
```



2. When the assembly is loaded from the DLL (complete and partial references below):

```

        declare namespace cs="clitype:MyManagedDLL.testClass?from=file:///
C:/Altova
        Projects/extFunctions/MyManagedDLL.dll;

        declare namespace cs="clitype:MyManagedDLL.testClass?
from=MyManagedDLL.dll;

```

### XSLT example

Here is a complete XSLT example that calls functions in system class `System.Math`:

```

<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions">
  <xsl:output method="xml" omit-xml-declaration="yes" />
  <xsl:template match="/">
    <math xmlns:math="clitype:System.Math">
      <sqrt><xsl:value-of select="math:Sqrt(9)"/></sqrt>
      <pi><xsl:value-of select="math:PI()"/></pi>
      <e><xsl:value-of select="math:E()"/></e>
      <pow><xsl:value-of select="math:Pow(math:PI(), math:E())"/></pow>
    </math>
  </xsl:template>
</xsl:stylesheet>

```

The namespace declaration on the element `math` associates the prefix `math:` with the URI `clitype:System.Math`. The `clitype:` beginning of the URI indicates that what follows identifies either a system class or a loaded class. The `math:` prefix in the XPath expressions associates the extension functions with the URI (and, by extension, the class) `System.Math`. The extension functions identify methods in the class `System.Math` and supply arguments where required.

### XQuery example

Here is an XQuery example fragment similar to the XSLT example above:

```

<math xmlns:math="clitype:System.Math">
  {math:Sqrt(9)}
</math>

```

As with the XSLT example above, the namespace declaration identifies the .NET class, in this case a system class. The XQuery expression identifies the method to be called and supplies the argument.

### .NET: Constructors

An extension function can be used to call a .NET constructor. All constructors are called with the pseudo-function `new()`. If there is more than one constructor for a class, then the constructor that



most closely matches the number of arguments supplied is selected. If no constructor is deemed to match the supplied argument/s, then a 'No constructor found' error is returned.

### Constructors that return XPath/XQuery datatypes

If the result of a .NET constructor call can be [implicitly converted to XPath/XQuery datatypes](#), then the .NET extension function will return a sequence that is an XPath/XQuery datatype.

### Constructors that return .NET objects

If the result of a .NET constructor call cannot be converted to a suitable XPath/XQuery datatype, then the constructor creates a wrapped .NET object with a type that is the name of the class returning that object. For example, if a constructor for the class `System.DateTime` is called (with `System.DateTime.new()`), then an object having a type `System.DateTime` is returned.

The lexical format of the returned object may not match the lexical format of a required XPath datatype. In such cases, the returned value would need to be: (i) converted to the lexical format of the required XPath datatype; and (ii) cast to the required XPath datatype.

There are three things that can be done with a .NET object created by a constructor:

- It can be used within a variable:  

```
<xsl:variable name="currentdate" select="date:new(2008, 4, 29)"
xmlns:date="clitype:System.DateTime" />
```
- It can be passed to an extension function (see [Instance Method and Instance Fields](#)):  

```
<xsl:value-of select="date:ToString(date:new(2008, 4, 29))"
xmlns:date="clitype:System.DateTime" />
```
- It can be converted to a string, number, or boolean:
- ```
<xsl:value-of select="xs:integer(data:get_Month(date:new(2008, 4, 29)))"
xmlns:date="clitype:System.DateTime" />
```

### .NET: Static Methods and Static Fields

A static method is called directly by its name and by supplying the arguments for the method. The name used in the call must exactly match a public static method in the class specified. If the method name and the number of arguments that were given in the function call matches more than one method in a class, then the types of the supplied arguments are evaluated for the best match. If a match cannot be found unambiguously, an error is reported.

**Note:** A field in a .NET class is considered to be a method without any argument. A property is called using the syntax `get_PropertyName()`.

### Examples

An XSLT example showing a call to a method with one argument (`System.Math.Sin(arg)`):

```
<xsl:value-of select="math:Sin(30)" xmlns:math="clitype:System.Math"/>
```



An XSLT example showing a call to a field (considered a method with no argument)

```
(System.Double.MaxValue());
<xsl:value-of select="double:MaxValue()" xmlns:double="clitype:System.Double"/>
```

An XSLT example showing a call to a property (syntax is `get_PropertyName()`)

```
(System.String());
<xsl:value-of select="string:get_Length('my string')"
xmlns:string="clitype:System.String"/>
```

An XQuery example showing a call to a method with one argument (`System.Math.Sin(arg)`):

```
<sin xmlns:math="clitype:System.Math">
  { math:Sin(30) }
</sin>
```

## .NET: Instance Methods and Instance Fields

An instance method has a .NET object passed to it as the first argument of the method call. This .NET object typically would be created by using an extension function (for example a constructor call) or a stylesheet parameter/variable. An XSLT example of this kind would be:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <xsl:template match="/">
    <xsl:variable name="releasedate"
      select="date:new(2008, 4, 29)"
      xmlns:date="clitype:System.DateTime"/>
    <doc>
      <date>
        <xsl:value-of select="date:ToString(date:new(2008, 4, 29))"
          xmlns:date="clitype:System.DateTime"/>
      </date>
      <date>
        <xsl:value-of select="date:ToString($releasedate)"
          xmlns:date="clitype:System.DateTime"/>
      </date>
    </doc>
  </xsl:template>
</xsl:stylesheet>
```

In the example above, a `System.DateTime` constructor (`new(2008, 4, 29)`) is used to create a .NET object of type `System.DateTime`. This object is created twice, once as the value of the variable `releasedate`, a second time as the first and only argument of the `System.DateTime.ToString()` method. The instance method `System.DateTime.ToString()` is called twice, both times with the `System.DateTime` constructor (`new(2008, 4, 29)`) as its first and only argument. In one of these instances, the variable `releasedate` is used to get the .NET object.



### Instance methods and instance fields

The difference between an instance method and an instance field is theoretical. In an instance method, a .NET object is directly passed as an argument; in an instance field, a parameter or variable is passed instead—though the parameter or variable may itself contain a .NET object. For example, in the example above, the variable `releasedate` contains a .NET object, and it is this variable that is passed as the argument of `ToString()` in the second `date` element constructor. Therefore, the `ToString()` instance in the first `date` element is an instance method while the second is considered to be an instance field. The result produced in both instances, however, is the same.

### Datatypes: XPath/XQuery to .NET

When a .NET extension function is used within an XPath/XQuery expression, the datatypes of the function's arguments are important for determining which one of multiple .NET methods having the same name is called.

In .NET, the following rules are followed:

- If there is more than one method with the same name in a class, then the methods available for selection are reduced to those that have the same number of arguments as the function call.
- The XPath/XQuery string, number, and boolean datatypes (*see list below*) are implicitly converted to a corresponding .NET datatype. If the supplied XPath/XQuery type can be converted to more than one .NET type (for example, `xs:integer`), then that .NET type is selected which is declared for the selected method. For example, if the .NET method being called is `fx(double)` and the supplied XPath/XQuery datatype is `xs:integer`, then `xs:integer` will be converted to .NET's `double` datatype.

The table below lists the implicit conversions of XPath/XQuery string, number, and boolean types to .NET datatypes.

<code>xs:string</code>	<code>StringValue</code> , <code>string</code>
<code>xs:boolean</code>	<code>BooleanValue</code> , <code>bool</code>
<code>xs:integer</code>	<code>IntegerValue</code> , <code>decimal</code> , <code>long</code> , <code>integer</code> , <code>short</code> , <code>byte</code> , <code>double</code> , <code>float</code>
<code>xs:float</code>	<code>FloatValue</code> , <code>float</code> , <code>double</code>
<code>xs:double</code>	<code>DoubleValue</code> , <code>double</code>
<code>xs:decimal</code>	<code>DecimalValue</code> , <code>decimal</code> , <code>double</code> , <code>float</code>

Subtypes of the XML Schema datatypes listed above (and which are used in XPath and XQuery) will also be converted to the .NET type/s corresponding to that subtype's ancestor type.



In some cases, it might not be possible to select the correct .NET method based on the supplied information. For example, consider the following case.

- The supplied argument is an `xs:untypedAtomic` value of 10 and it is intended for the method `mymethod(float)`.
- However, there is another method in the class which takes an argument of another datatype: `mymethod(double)`.
- Since the method names are the same and the supplied type (`xs:untypedAtomic`) could be converted correctly to either `float` or `double`, it is possible that `xs:untypedAtomic` is converted to `double` instead of `float`.
- Consequently the method selected will not be the required method and might not produce the expected result. To work around this, you can create a user-defined method with a different name and use this method.

Types that are not covered in the list above (for example `xs:date`) will not be converted and will generate an error.

### Datatypes: .NET to XPath/XQuery

When a .NET method returns a value and the datatype of the value is a string, numeric or boolean type, then it is converted to the corresponding XPath/XQuery type. For example, .NET's `decimal` datatype is converted to `xsd:decimal`.

When a .NET object or a datatype other than string, numeric or boolean is returned, you can ensure conversion to the required XPath/XQuery type by first using a .NET method (for example `System.DateTime.ToString()`) to convert the .NET object to a string. In XPath/XQuery, the string can be modified to fit the lexical representation of the required type and then converted to the required type (for example, by using the `cast as expression`).

### MSXSL Scripts for XSLT

The `<msxsl:script>` element contains user-defined functions and variables that can be called from within XPath expressions in the XSLT stylesheet. The `<msxsl:script>` is a top-level element, that is, it must be a child element of `<xsl:stylesheet>` or `<xsl:transform>`.

The `<msxsl:script>` element must be in the namespace `urn:schemas-microsoft-com:xslt` (see *example below*).

### Scripting language and namespace

The scripting language used within the block is specified in the `<msxsl:script>` element's `language` attribute and the namespace to be used for function calls from XPath expressions is identified with the `implements-prefix` attribute (see *below*).

```
<msxsl:script language="scripting-language" implements-prefix="user-namespace-prefix">
```



```

function-1 or variable-1
...
function-n or variable-n

</msxsl:script>

```

The `<msxsl:script>` element interacts with the Windows Scripting Runtime, so only languages that are installed on your machine may be used within the `<msxsl:script>` element. **The .NET Framework 2.0 platform or higher must be installed for MSXSL scripts to be used.** Consequently, the .NET scripting languages can be used within the `<msxsl:script>` element.

The `language` attribute accepts the same values as the `language` attribute on the HTML `<script>` element. If the `language` attribute is not specified, then Microsoft JScript is assumed as the default.

The `implements-prefix` attribute takes a value that is a prefix of a declared in-scope namespace. This namespace typically will be a user namespace that has been reserved for a function library. All functions and variables defined within the `<msxsl:script>` element will be in the namespace identified by the prefix specified in the `implements-prefix` attribute. When a function is called from within an XPath expression, the fully qualified function name must be in the same namespace as the function definition.

## Example

Here is an example of a complete XSLT stylesheet that uses a function defined within a `<msxsl:script>` element.

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:user="http://mycompany.com/mynamespace">

  <msxsl:script language="VBScript" implements-prefix="user">
    <![CDATA[
      ' Input: A currency value: the wholesale price
      ' Returns: The retail price: the input value plus 20% margin,
      ' rounded to the nearest cent
      dim a as integer = 13
      Function AddMargin(WholesalePrice) as integer
        AddMargin = WholesalePrice * 1.2 + a
      End Function
    ]>
  </msxsl:script>

  <xsl:template match="/">
    <html>
      <body>
        <p>
          <b>Total Retail Price =
            $<xsl:value-of select="user:AddMargin(50)"/>
          </b>
        </p>
      </body>
    </html>
  </template>

```



```
<br/>
<b>Total Wholesale Price =
  $<xsl:value-of select="50"/>
</b>
</p>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

## Datatypes

The values of parameters passed into and out of the script block are limited to XPath datatypes. This restriction does not apply to data passed among functions and variables within the script block.

## Assemblies

An assembly can be imported into the script by using the `msxsl:assembly` element. The assembly is identified via a name or a URI. The assembly is imported when the stylesheet is compiled. Here is a simple representation of how the `msxsl:assembly` element is to be used.

```
<msxsl:script>
  <msxsl:assembly name="myAssembly.assemblyName" />
  <msxsl:assembly href="pathToAssembly" />

  ...

</msxsl:script>
```

The assembly name can be a full name, such as:

```
"system.Math, Version=3.1.4500.1 Culture=neutral
PublicKeyToken=a46b3f648229c514"
```

or a short name, such as `"myAssembly.Draw"`.

## Namespaces

Namespaces can be declared with the `msxsl:using` element. This enables assembly classes to be written in the script without their namespaces, thus saving you some tedious typing. Here is how the `msxsl:using` element is used so as to declare namespaces.

```
<msxsl:script>
  <msxsl:using namespace="myAssemblyNS.NamespaceName" />

  ...

</msxsl:script>
```

The value of the `namespace` attribute is the name of the namespace.







## 11.2 Technical Data

This section contains useful background information on the technical aspects of your software. It is organized into the following sections:

- [OS and Memory Requirements](#)
- [Altova XML Validator](#)
- [Altova XSLT and XQuery Engines](#)
- [Unicode Support](#)
- [Internet Usage](#)

### 11.2.1 OS and Memory Requirements

#### Operating System

Altova software applications are available for the following platforms:

- Windows 7 SP1 with Platform Update, Windows 8, Windows 10
- Windows Server 2008 R2 SP1 with Platform Update or newer

#### Memory

Since the software is written in C++ it does not require the overhead of a Java Runtime Environment and typically requires less memory than comparable Java-based applications. However, each document is loaded fully into memory so as to parse it completely and to improve viewing and editing speed. The memory requirement increases with the size of the document.

Memory requirements are also influenced by the unlimited Undo history. When repeatedly cutting and pasting large selections in large documents, available memory can rapidly be depleted.

### 11.2.2 Altova XML Validator

When opening any XML document, the application uses its built-in XML validator to check for well-formedness, validate the document against a schema (if specified), and build trees and infosets. The XML validator is also used to provide intelligent editing help while you edit documents and to dynamically display any validation error that may occur.

The built-in XML validator implements the Final Recommendation of the W3C's XML Schema 1.0 and 1.1 specification. New developments recommended by the W3C's XML Schema Working Group are continuously being incorporated in the XML validator, so that Altova products give you a state-of-the-art development environment.

### 11.2.3 Altova XSLT and XQuery Engines

Altova products use the Altova XSLT 1.0, 2.0, and 3.0 Engines and the Altova XQuery 1.0 and 3.1 Engines. Documentation about implementation-specific behavior for each engine is in the appendices of the documentation (Engine Information), should that engine be used in the product.

**Note:** Altova MapForce generates code using the XSLT 1.0, 2.0 and XQuery 1.0 engines.



## 11.2.4 Unicode Support

Altova's XML products provide full Unicode support. To edit an XML document, you will also need a font that supports the Unicode characters being used by that document.

Please note that most fonts only contain a very specific subset of the entire Unicode range and are therefore typically targeted at the corresponding writing system. If some text appears garbled, the reason could be that the font you have selected does not contain the required glyphs. So it is useful to have a font that covers the entire Unicode range, especially when editing XML documents in different languages or writing systems. A typical Unicode font found on Windows PCs is Arial Unicode MS.

In the `/Examples` folder of your application folder you will find an XHTML file called `UnicodeUTF-8.html` that contains the following sentence in a number of different languages and writing systems:

- *When the world wants to talk, it speaks Unicode*
- *Wenn die Welt miteinander spricht, spricht sie Unicode*
- 世界的に話すなら、Unicode です。

Opening this XHTML file will give you a quick impression of Unicode's possibilities and also indicate what writing systems are supported by the fonts available on your PC.

## 11.2.5 Internet Usage

Altova applications will initiate Internet connections on your behalf in the following situations:

- If you click the "Request evaluation key-code" in the Registration dialog (**Help | Software Activation**), the three fields in the registration dialog box are transferred to our web server by means of a regular http (port 80) connection and the free evaluation key-code is sent back to the customer via regular SMTP e-mail.
- In some Altova products, you can open a file over the Internet (**File | Open | Switch to URL**). In this case, the document is retrieved using one of the following protocol methods and connections: HTTP (normally port 80), FTP (normally port 20/21), HTTPS (normally port 443). You could also run an HTTP server on port 8080. (In the URL dialog, specify the port after the server name and a colon.)
- If you open an XML document that refers to an XML Schema or DTD and the document is specified through a URL, the referenced schema document is also retrieved through a HTTP connection (port 80) or another protocol specified in the URL (see Point 2 above). A schema document will also be retrieved when an XML file is validated. Note that validation might happen automatically upon opening a document if you have instructed the application to do this (in the File tab of the Options dialog (**Tools | Options**)).
- In Altova applications using WSDL and SOAP, web service connections are defined by the WSDL documents.
- If you are using the **Send by Mail** command (**File | Send by Mail**) in XMLSpy, the current selection or file is sent by means of any MAPI-compliant mail program installed on the user's PC.
- As part of Software Activation and LiveUpdate as further described in the Altova Software License Agreement.



## 11.3 License Information

This section contains:

- Information about the [distribution of this software product](#)
- Information about [software activation and license metering](#)
- Information about the [intellectual property rights](#) related to this software product
- The [End-User License Agreement](#) governing the use of this software product

Please read this information carefully. It is binding upon you since you agreed to these terms when you installed this software product.

### 11.3.1 Electronic Software Distribution

This product is available through electronic software distribution, a distribution method that provides the following unique benefits:

- You can evaluate the software free-of-charge before making a purchasing decision.
- Once you decide to buy the software, you can place your order online at the [Altova website](#) and immediately get a fully licensed product within minutes.
- When you place an online order, you always get the latest version of our software.
- The product package includes a comprehensive integrated onscreen help system. The latest version of the user manual is available at [www.altova.com](#) (i) in HTML format for online browsing, and (ii) in PDF format for download (and to print if you prefer to have the documentation on paper).

#### 30-day evaluation period

After downloading this product, you can evaluate it for a period of up to 30 days free of charge. About 20 days into this evaluation period, the software will start to remind you that it has not yet been licensed. The reminder message will be displayed once each time you start the application. If you would like to continue using the program after the 30-day evaluation period, you have to purchase an [Altova Software License Agreement](#), which is delivered in the form of a key-code that you enter into the Software Activation dialog to unlock the product. You can purchase your license at the online shop at the [Altova website](#).

#### Helping Others within Your Organization to Evaluate the Software

If you wish to distribute the evaluation version within your company network, or if you plan to use it on a PC that is not connected to the Internet, you may only distribute the Setup programs, provided that they are not modified in any way. Any person that accesses the software installer that you have provided, must request their own 30-day evaluation license key code and after expiration of their evaluation period, must also purchase a license in order to be able to continue using the product.

For further details, please refer to the [Altova Software License Agreement](#) at the end of this section.



### 11.3.2 Software Activation and License Metering

As part of Altova's Software Activation, the software may use your internal network and Internet connection for the purpose of transmitting license-related data at the time of installation, registration, use, or update to an Altova-operated license server and validating the authenticity of the license-related data in order to protect Altova against unlicensed or illegal use of the software and to improve customer service. Activation is based on the exchange of license related data such as operating system, IP address, date/time, software version, and computer name, along with other information between your computer and an Altova license server.

Your Altova product has a built-in license metering module that further helps you avoid any unintentional violation of the End User License Agreement. Your product is licensed either as a single-user or multi-user installation, and the license-metering module makes sure that no more than the licensed number of users use the application concurrently.

This license-metering technology uses your local area network (LAN) to communicate between instances of the application running on different computers.

#### Single license

When the application starts up, as part of the license metering process, the software sends a short broadcast datagram to find any other instance of the product running on another computer in the same network segment. If it doesn't get any response, it will open a port for listening to other instances of the application.

#### Multi license

If more than one instance of the application is used within the same LAN, these instances will briefly communicate with each other on startup. These instances exchange key-codes in order to help you to better determine that the number of concurrent licenses purchased is not accidentally violated. This is the same kind of license metering technology that is common in the Unix world and with a number of database development tools. It allows Altova customers to purchase reasonably-priced concurrent-use multi-user licenses.

We have also designed the applications so that they send few and small network packets so as to not put a burden on your network. The TCP/IP ports (2799) used by your Altova product are officially registered with the IANA (see [the IANA website \(http://www.iana.org/\)](http://www.iana.org/) for details) and our license-metering module is tested and proven technology.

If you are using a firewall, you may notice communications on port 2799 between the computers that are running Altova products. You are, of course, free to block such traffic between different groups in your organization, as long as you can ensure by other means, that your license agreement is not violated.

You will also notice that, if you are online, your Altova product contains many useful functions; these are unrelated to the license-metering technology.



### 11.3.3 Intellectual Property Rights

The Altova Software and any copies that you are authorized by Altova to make are the intellectual property of and are owned by Altova and its suppliers. The structure, organization and code of the Software are the valuable trade secrets and confidential information of Altova and its suppliers. The Software is protected by copyright, including without limitation by United States Copyright Law, international treaty provisions and applicable laws in the country in which it is being used. Altova retains the ownership of all patents, copyrights, trade secrets, trademarks and other intellectual property rights pertaining to the Software, and that Altova's ownership rights extend to any images, photographs, animations, videos, audio, music, text and "applets" incorporated into the Software and all accompanying printed materials. Notifications of claimed copyright infringement should be sent to Altova's copyright agent as further provided on the Altova Web Site.

Altova software contains certain Third Party Software that is also protected by intellectual property laws, including without limitation applicable copyright laws as described in detail at [http://www.altova.com/legal\\_3rdparty.html](http://www.altova.com/legal_3rdparty.html).

All other names or trademarks are the property of their respective owners.

### 11.3.4 Altova End User License Agreement

- The Altova End User License Agreement is available here: <http://www.altova.com/eula>
- Altova's Privacy Policy is available here: <http://www.altova.com/privacy>



# Chapter 12

---

## Glossary



## 12 Glossary

The glossary section includes the list of terms pertaining to MapForce.



## 12.1 C

### Component

In MapForce, the term "component" is what represents visually the structure (schema) of your data, or how data is to be transformed (functions). Components are the central building pieces of any [mapping](#). On the mapping area, components appear as rectangles. The following are examples of MapForce components:

- Constants
- Filters
- Conditions
- Function components
- EDI documents (UN/EDIFACT, ANSI X12, HL7)
- Excel 2007+ files
- Simple [input components](#)
- Simple [output components](#)
- XML Schemas and DTDs

### Connection

A connection is a line that you can draw between two [connectors](#). By drawing connections, you instruct MapForce to transform data in a specific way (for example, read data from an XML document and write it to another XML document).

### Connector

A connector is a small triangle displayed on the left or right side of a [component](#). The connectors displayed on the left of a component provide data entry points *to that component*. The connectors displayed on the right of a component provide data exit points *from that component*.



## 12.2 F

### **Fixed Length Field (FLF)**

A common text format where data is conventionally separated into fields which have a fixed length (for example, the first 5 characters of every row represent a transaction ID, and the next 20 characters represent a transaction description).

### **FlexText**

FlexText is a module in MapForce Enterprise Edition which enables you to convert data from non-standard or legacy text files of high complexity to other formats supported by MapForce, and vice versa.



## 12.3 G

### **Global Resources**

Altova Global Resources represent a way to refer to files, folders, or databases so as to make these resources reusable, configurable and available across multiple Altova applications.



## 12.4 I

### **Input component**

An input component is a MapForce [component](#) that enables you to pass simple values to a mapping. Input components are commonly used to pass file names or other string values to a mapping at runtime. Input components should not be confused with [source components](#).



## 12.5 J

### **Join component**

A Join component is a MapForce [component](#) which enables joining two or more structures on the mapping based on custom-defined conditions. It returns the association (joined set) of items that satisfy the condition. Joins are particularly useful to combine data from two structures which share a common field (such as an identity).



## 12.6 M

### MapForce

MapForce is a Windows-based, multi-purpose IDE (integrated development environment) that enables you to transform data from one format to another, or from one schema to another, by means of a visual, "drag-and-drop" -style graphical user interface that does not require writing any program code. In fact, MapForce generates for you the program code which performs the actual data transformation (or data mapping). When you prefer not to generate program code, you can just run the transformation using the MapForce built-in transformation language (available in the MapForce Professional or Enterprise Editions).

### Mapping

A MapForce mapping design (or simply "mapping") is the visual representation of how data is to be transformed from one format to another. A mapping consists of [components](#) that you add to the MapForce mapping area in order to create your data transformations (for example, convert XML documents from one schema to another). A valid mapping consists of one or several [source components](#) connected to one or several [target components](#). You can run a mapping and preview its result directly in MapForce. You can generate code and execute it externally. You can also compile a mapping to a MapForce execution file and automate mapping execution using MapForce Server or FlowForce Server. MapForce saves mappings as files with .mfd extension.

### MFF

The file name extension of MapForce function files.

### MFD

The file name extension of MapForce design documents ([mappings](#)).



## 12.7 O

### **Output component**

An output component (or "simple output") is a MapForce [component](#) which enables you to return a string value from the mapping. Output components represent just one possible type of [target components](#), but should not be confused with the latter.



## 12.8 P

### **parent-context**

**parent-context** is an optional argument in some MapForce core aggregation functions such as **min**, **max**, **avg**, **count**. In a source component which has multiple hierarchical sequences, the parent context determines the set of nodes on which the function should operate.



## 12.9 S

### **Source component**

A source component is a [component](#) from which MapForce reads data. When you run the [mapping](#), MapForce reads the data supplied by the connector of the source component, converts it to the required type, and sends it to the connector of the [target component](#).



## 12.10 T

### **Target component**

A target component is a [component](#) to which MapForce writes data. When you run the [mapping](#), a target component instructs MapForce to either generate a file (or multiple files) or output the result as a string value for further processing in an external program. A target component is the opposite of a [source component](#).



# Index

■

**.NET extension functions,**

- constructors, 478
- datatype conversions, .NET to XPath/XQuery, 482
- datatype conversions, XPath/XQuery to .NET, 481
- for XSLT and XQuery, 476
- instance methods, instance fields, 480
- overview, 476
- static methods, static fields, 479

## A

**A to Z,**

- sort component, 168

**abs,**

- as MapForce function (in xpath2 | numeric functions), 354

**add,**

- as MapForce function (in core | math functions), 320

**Altova Engines,**

- in Altova products, 486

**Altova extensions,**

- chart functions (see chart functions), 420

**Altova XML Parser,**

- about, 486

**Any,**

- xs:any, 234

**ATTLIST,**

- DTD namespace URIs, 226

**auto-number,**

- as MapForce function (in core | generator functions), 314

**avg,**

- as MapForce function (in core | aggregate functions), 301

## B

**Background Information, 486****base-uri,**

- as MapForce function (in xpath2 | accessors library), 347

**Bool,**

- output if false, 265

**boolean,**

- as MapForce function (in core | conversion functions), 305

**Built-in engine,**

- definition, 69
- using, 69

## C

**CDATA, 233****ceiling,**

- as MapForce function (in core | math functions), 320

**char-from-code,**

- as MapForce function (in core | string functions), 338

**Code,**

- inline functions & code size, 259

**Code point,**

- collation, 168

**code-from-char,**

- as MapForce function (in core | string functions), 339

**Collation,**

- locale collation, 168
- sort component, 168
- unicode code point, 168

**Comments,**

- Adding to target files, 231

**Complex,**

- function - inline, 259
- User-defined complex input, 271
- User-defined complex output, 276
- User-defined function, 270, 276

**Complex type,**

- sorting, 168

**Component,**

- as application menu, 397
- definition of, 493
- deleted items, 106
- sort data, 168

**Components,**

- adding to the mapping, 65
- aligning, 89
- changing settings, 90
- overview, 87
- processing sequence, 209
- searching, 88



**concat,**  
as MapForce function (in core | string functions), 339

**Connection,**  
as application menu, 398  
definition of, 493

**Connections,**  
moving to a different component, 101  
preserving on root element change, 101

**Connector,**  
definition of, 493

**Consolidating data,**  
merging XML files, 238

**Constants,**  
adding to the mapping, 248

**contains,**  
as MapForce function (in core | string functions), 339

**Copy all,**  
mapping method, 119

**Copyright information, 488**

**count,**  
as MapForce function (in core | aggregate functions), 301

**current,**  
as MapForce function (in xslt | xslt functions library), 359

**current-date,**  
as MapForce function (in xpath2 | context functions), 349

**current-dateTime,**  
as MapForce function (in xpath2 | context functions), 349

**current-time,**  
as MapForce function (in xpath2 | context functions), 349

## D

**Default,**  
input value, 265

**default-collation,**  
as MapForce function (in xpath2 | context functions), 349

**Delete,**  
deletions - missing items, 106

**Derived types,**  
mapping to/from, 227

**Digital signature,**  
creating in XML output, 222

**distinct-values,**  
as MapForce function (in core | sequence functions), 326

**Distribution,**  
of Altova's software products, 488, 490

**divide,**  
as MapForce function (in core | math functions), 321

**document,**  
as MapForce function (in xslt | xslt functions library), 359

**DoTransform.bat,**  
execute with RaptorXML Server, 365

**DTD,**  
source and target, 226

**Duplicate input, 39**  
adding, 397

## E

**Edit,**  
as application menu, 394

**Element,**  
recursive element in XML Schema, 280

**element-available,**  
as MapForce function (in xslt | xslt functions library), 359

**Encoding settings,**  
in XML output, 222

**End User License Agreement, 488, 490**

**equal,**  
as MapForce function (in core | logical functions), 317

**equal-or-greater,**  
as MapForce function (in core | logical functions), 317

**equal-or-less,**  
as MapForce function (in core | logical functions), 318

**Evaluation period,**  
of Altova's software products, 488, 490

**Example,**  
recursive user-defined mapping, 280

**exists,**  
as MapForce function (in core | sequence functions), 327

**Extension functions for XSLT and XQuery, 467**

**Extension Functions in .NET for XSLT and XQuery,**  
see under .NET extension functions, 476

**Extension Functions in Java for XSLT and XQuery,**  
see under Java extension functions, 467

**Extension Functions in MSXSL scripts, 482**

## F

**false,**



**false**,  
as MapForce function (in xpath2 | boolean functions), 348

**File**,  
as application menu, 391  
as button on a component, 90  
as button on components, 141

**File names**,  
supplying as mapping input parameters, 145

**File paths**,  
fixing broken references, 116  
in generated code, 117  
relative versus absolute, 114, 117

**File/String**,  
as button on a component, 90  
as button on components, 141

**File: (default)**,  
as name of root node, 141

**File: <dynamic>**,  
as name of root node, 141

**Filter**,  
merging XML files, 238

**Filtering**,  
data from components, 174  
database tables, 174

**Filters**,  
adding to the mapping, 174

**first-items**,  
as MapForce function (in core | sequence functions), 328

**FlexText**,  
definition of, 494

**FLF**,  
definition of, 494

**floor**,  
as MapForce function (in core | math functions), 321

**format-date**,  
as MapForce function (in core | conversion functions), 305

**format-dateTime**,  
as MapForce function (in core | conversion functions), 306

**format-number**,  
as MapForce function (in core | conversion functions), 309

**format-time**,  
as MapForce function (in core | conversion functions), 311

**Function, 259**  
as application menu, 399  
complex - inline, 259  
inline, 259  
nested user-defined, 265  
standard user-defined function, 261  
user-defined function, 282  
user-defined look-up function, 261

**function-available**,  
as MapForce function (in xslt | xslt functions library), 360

**Functions**,  
adding as mapping components, 247  
adding parameters to, 251  
deleting parameters from, 251  
finding in the Libraries window, 249  
finding occurrences in active mapping, 249  
viewing the argument data type of, 250  
viewing the description of, 250

## G

**Generate**,  
code & inline functions, 259

**generate-id**,  
as MapForce function (in xslt | xslt functions library), 360

**generate-sequence**,  
as MapForce function (in core | sequence functions), 328

**get-fileext**,  
as MapForce function (in core | file path functions), 312

**get-folder**,  
as MapForce function (in core | file path functions), 312

**Global Resources**,  
creating, 373  
examples of usage, 375, 377  
introduction to, 373

**greater**,  
as MapForce function (in core | logical functions), 318

**group-adjacent**,  
as MapForce function (in core | sequence functions), 328

**group-by**,  
as MapForce function (in core | sequence functions), 329

**group-ending-with**,  
as MapForce function (in core | sequence functions), 330

**group-into-blocks**,  
as MapForce function (in core | sequence functions), 330

**group-starting-with**,  
as MapForce function (in core | sequence functions), 330



# H

## Health Level 7,

example, 243

## Help,

as application menu, 405

## HL7 2.6 to 3.x,

example, 243

# I

## If-Else conditions,

adding to the mapping, 174

## implicit-timezone,

as MapForce function (in xpath2 | context functions), 349

## Inline,

functions and code size, 259

## Inline / Standard,

user-defined functions, 259

## Input, 265

default value, 265

optional parameters, 265

## Input component,

definition of, 496

## Insert,

as application menu, 395

## Instance,

changing the path reference to, 114

## Internet usage,

in Altova products, 487

## is-xsi-nil,

as MapForce function (in core | node functions), 323

## Item,

missing, 106

## item-at,

as MapForce function (in core | sequence functions), 331

## items-from-till,

as MapForce function (in core | sequence functions), 331

# J

## Java extension functions,

constructors, 473

datatype conversions, Java to Xpath/XQuery, 476

datatype conversions, XPath/XQuery to Java, 475

for XSLT and XQuery, 467

instance methods, instance fields, 474

overview, 467

static methods, static fields, 473

user-defined class files, 469

user-defined JAR files, 472

# K

## Keeping data,

when using value-map, 183

## Keeping data unchanged,

passing through a value-map, 183

## Key,

sort key, 168

# L

## last,

as MapForce function (in xpath2 | context functions), 349

## last-items,

as MapForce function (in core | sequence functions), 332

## Legal information, 488

## less,

as MapForce function (in core | logical functions), 318

## Libraries window,

finding functions in, 249

## License, 490

information about, 488

## License metering,

in Altova products, 489

## Locale collation, 168

## local-name-from-QName,

as MapForce function (in lang | QName functions), 326

## logical-and,

as MapForce function (in core | logical functions), 318

## logical-not,

as MapForce function (in core | logical functions), 319

## logical-or,

as MapForce function (in core | logical functions), 319

## Lookup table,



**Lookup table,**  
properties, 185  
value map table, 180

## M

**main-mfd-filepath,**  
as MapForce function (in core | file path functions), 312

**MapForce,**  
basic concepts, 18  
overview, 12

**MapForce samples,**  
location on disk, 26

**Mapping,**  
creating, 65  
definition of, 498  
processing sequence, 209  
source driven - mixed content, 119  
validating, 70

**Mapping input,**  
supplying custom file name as, 145  
Supplying multiple files as, 141, 143, 144

**Mapping methods,**  
standard, 119  
standard / mixed / copy all, 119  
target-driven, 119

**Mapping output,**  
Generating multiple files as, 141, 144

**Marked items,**  
missing items, 106

**max,**  
as MapForce function (in core | aggregate functions), 302

**max-string,**  
as MapForce function (in core | aggregate functions), 302

**Memory requirements, 486**

**Merging,**  
XML files, 238

**mfd,**  
as file extension, 498

**mfd-filepath,**  
as MapForce function (in core | file path functions), 313

**mff,**  
as file extension, 498

**mfp,**  
as file extension, 498

**mft,**

as file extension, 498

**Microsoft SharePoint Server,**  
adding files as components from, 66

**min,**  
as MapForce function (in core | aggregate functions), 303

**min-string,**  
as MapForce function (in core | aggregate functions), 303

**Missing items, 106**

**Mixed, 119**  
content mapping, 119  
content mapping example, 125  
content mapping method, 119  
source-driven mapping, 119

**Mixed content,**  
Mapping, 126

**modulus,**  
as MapForce function (in core | math functions), 321

**msxsl:script, 482**

**Multiple source,**  
to single target, 238

**multiply,**  
as MapForce function (in core | math functions), 322

## N

**Namespace URI,**  
DTD, 226

**Namespace URIs,**  
and QNames, 229

**Namespaces,**  
and wildcards (xs:any), 234  
declaring custom, 240

**namespace-uri-form-QName,**  
as MapForce function (in lang | QName functions), 326

**Nested,**  
user-defined functions, 265

**nillable,**  
as attribute in XML schema, 229

**Node names,**  
mapping data from/to, 188

**node-name,**  
as MapForce function (in core | node functions), 323  
as MapForce function (in xpath2 | accessors library), 347

**node-name function,**  
alternatives to using, 188

**normalize-space,**



**normalize-space,**  
as MapForce function (in core | string functions), 339

**not-equal,**  
as MapForce function (in core | logical functions), 319

**not-exists,**  
as MapForce function (in core | sequence functions), 332

**number,**  
as MapForce function (in core | conversion functions), 311

## O

**Optional,**  
input parameters, 265

**Order,**  
components are processed, 209

**Ordering data,**  
sort component, 168

**OS,**  
for Altova products, 486

**Output, 265**  
as application menu, 400  
parameter, 265  
previewing, 72  
saving, 72  
user-defined if bool = false, 265  
validating, 71

**Output component,**  
definition of, 499

## P

**Parameter, 265**  
optional, 265  
output, 265

**parent-context,**  
definition of, 500

**Parser,**  
built into Altova products, 486

**Passing through data,**  
unchanged through value-map, 183

**Paths in generated code,**  
making absolute, 84

**Platforms,**  
for Altova products, 486

**position,**  
as MapForce function (in core | sequence functions), 333

**Priority Context,**  
setting on functions, 212

**Processing Instructions,**  
Adding to target files, 231

**Processing Instructions and Comments,**  
mapping, 120

**Processing sequence,**  
of components in a mapping, 209

**Properties,**  
value map table, 185

## Q

**QName,**  
as MapForce function (in lang | QName functions), 325

**QName support, 229**

**Question mark,**  
missing items, 106

## R

**RaptorXML Server,**  
executing a transformation, 365

**Recursive,**  
calls in functions, 259  
user-defined function, 282  
user-defined mapping, 280

**Reference, 390**

**Regular expressions,**  
as parameter to the "match-pattern" function, 297  
as parameter to the "tokenize-regexp" function, 297

**remove-fileext,**  
as MapForce function (in core | file path functions), 313

**remove-folder,**  
as MapForce function (in core | file path functions), 313

**replace-fileext,**  
as MapForce function (in core | file path functions), 313

**replicate-item,**  
as MapForce function (in core | sequence functions), 336

**replicate-sequence,**  
as MapForce function (in core | sequence functions), 337

**resolve-filepath,**



**resolve-filepath**,  
as MapForce function (in core | file path functions), 314

**resolve-uri**,  
as MapForce function (in xpath2 | anyURI functions), 347

**Retaining data**,  
passing through vlaue-map, 183

**round**,  
as MapForce function (in core | math functions), 322

**round-half-to-even**,  
as MapForce function (in xpath2 | numeric functions), 354

**round-precision**,  
as MapForce function (in core | math functions), 322

## S

**Schema**,  
and XML mapping, 221  
changing the path reference to, 114  
generating for an XML file, 221  
recursive elements, 280

**Scripts in XSLT/XQuery**,  
see under Extension functions, 467

**Search**,  
functions in the Libraries window, 249  
items within mapping components, 88

**Section**,  
CDATA, 233

**Sequence**,  
of processing components, 209

**set-empty**,  
as MapForce function (in core | sequence functions), 338

**set-xsi-nil**,  
as MapForce function (in core | node functions), 324

**Simple type**,  
sorting, 168

**Single target**,  
multiple sources, 238

**skip-first-items**,  
as MapForce function (in core | sequence functions), 338

**Software product license**, 490

**Sort**,  
sort component, 168

**Sort key**,  
sort component, 168

**Sort order**,  
changing, 168

**Source component**,  
definition of, 501

**Source-driven**,  
- mixed content mapping, 119

**Source-driven connections**,  
as opposed to standard (target-driven) connections, 126

**SQLite**,  
changing database path to absolute in generated code, 117

**Standard**,  
mapping method, 119

**starts-with**,  
as MapForce function (in core | string functions), 340

**static-node-annotation**,  
as MapForce function (in core | node functions), 324

**static-node-name**,  
as MapForce function (in core | node functions), 325

**string**,  
as MapForce function (in core | conversion functions), 312  
as MapForce function (in xpath2 | accessors library), 347

**string-join**,  
as MapForce function (in core | aggregate functions), 304

**string-length**,  
as MapForce function (in core | string functions), 340

**substitute-missing**,  
as MapForce function (in core | sequence functions), 338

**substitute-missing-with-xsi-nil**,  
as MapForce function (in core | node functions), 325

**substring**,  
as MapForce function (in core | string functions), 340

**substring-after**,  
as MapForce function (in core | string functions), 340

**substring-before**,  
as MapForce function (in core | string functions), 341

**subtract**,  
as MapForce function (in core | math functions), 322

**sum**,  
as MapForce function (in core | aggregate functions), 304

**system-property**,  
as MapForce function (in xslt | xslt functions library), 360

## T

**Table**,  
lookup - value map, 180

**Table data**,  
sorting, 168



**Target component,**

definition of, 502

**Target-driven connections,**

as opposed to source-driven connections, 126

**Target-driven mapping, 119****Technical Information, 486****tokenize,**

as MapForce function (in core | string functions), 341

**tokenize-by-length,**

as MapForce function (in core | string functions), 343

**tokenize-regex,**

as MapForce function (in core | string functions), 345

**Tools,**

as application menu, 403

**Transform,**

input data - value map, 180

**Transformation language,**

selecting, 69

**Transformations,**

RaptorXML Server, 365

**translate (in core | string functions),**

as MapForce function, 346

**true,**

as MapForce function (in xpath2 | boolean functions), 348

**Types,**

derived types - xsi:type, 227

## U

**Unicode,**

code point collation, 168

**Unicode support,**

in Altova products, 487

**unparsed-entity-uri,**

as MapForce function (in xslt | xslt functions library), 361

**URI,**

in DTDs, 226

**URIs,**

and QNames, 229

**URL,**

adding files as components from, 66

**User defined, 265**

complex input, 271

complex output, 276

function - inline / standard, 259

function - standard, 261

functions - complex, 270, 276

look-up functions, 261

nested functions, 265

output if bool = false, 265

**user-defined function,**

recursive, 282

**User-defined functions,**

creating, 252

deleting, 252

importing, 252

influencing the parameter order, 252

opening, 252

reusing, 252

## V

**Validate,**

mapping design, 70

mapping output, 71

**Validator,**

in Altova products, 486

**Value,**

default, 265

**Value-Map,**

lookup table, 180

lookup table - properties, 185

passing data unchanged, 183

**Variables,**

adding to the mapping, 160

changing the scope of, 163

examples of use, 165

introduction to, 159

**View,**

as application menu, 401

## W

**WebDAV Server,**

adding files as components from, 66

**Wildcards,**

xs:any - xs:any Attribute, 234

**Windows,**

support for Altova products, 486



# X

**XML declaration,**

- suppressing from output, 222

**XML files,**

- generate from single XML source, 146

**XML output,**

- changing encoding settings, 222
- changing instance file name, 222
- changing schema, 222
- creating digital signature, 222

**XML Parser,**

- about, 486

**XML to XML, 221****XQuery,**

- Extension functions, 467

**XQuery processor,**

- in Altova products, 486

**xs:any (xs:anyAttribute), 234****xsi:nil,**

- as attribute in XML instance, 229

**xsi:type,**

- mapping to derived types, 227

**XSLT,**

- adding custom functions, 291
- Extension functions, 467
- previewing the generated code, 82
- removing custom functions, 291
- template namespace, 291

**XSLT processors,**

- in Altova products, 486

# Z

**Z to A,**

- sort component, 168