Altova MapForce 2023 Basic Edition

ALTOVA®
MapForce®

User & Reference Manual

Altova MapForce 2023 Basic Edition
User & Reference Manual

Published: 2022

Table of Contents

1 Introduction 10

1.1 New Features 11
- 1.1.1 Version 2023 11
- 1.1.2 Version 2022 11
- 1.1.3 Version 2021 12
- 1.1.4 Version 2020 12
- 1.1.5 Version 2019 13

1.2 What Is MapForce? 14
- 1.2.1 Mapping: Sources and Targets 15
- 1.2.2 Transformation Languages 16
- 1.2.3 Mapping Scenarios 17
- 1.2.4 Integration with Altova Products 19

1.3 User Interface Overview 21
- 1.3.1 Bars 22
- 1.3.2 Windows 22
- 1.3.3 Messages Window 26
- 1.3.4 Panes 27

2 Tutorials 31

2.1 One Source to One Target 32
- 2.1.1 Create and Save Design 33
- 2.1.2 Add Source Component 34
- 2.1.3 Add Target Component 35
- 2.1.4 Connect Source and Target 37
- 2.1.5 Preview Mapping Result 40

2.2 Multiple Sources to One Target 42
- 2.2.1 Prepare Source Files 43
- 2.2.2 Add Target Component 44
- 2.2.3 Verify and Set Input/Output Files 45

2.2.4 Connect Sources and Target ... 46
2.3 Chained Mapping ... 49
2.3.1 Prepare Mapping Design ... 49
2.3.2 Configure Second Target ... 50
2.3.3 Connect Targets ... 52
2.3.4 Filter Data ... 52
2.3.5 Preview and Save Output ... 54
2.4 Multiple Sources to Multiple Targets ... 56
2.4.1 Prepare Mapping Design ... 58
2.4.2 Configure Input ... 59
2.4.3 Configure Output Part 1 ... 59
2.4.4 Configure Output Part 2 ... 62

3 Mapping Fundamentals 65

3.1 Components ... 67
3.1.1 Add Components ... 69
3.1.2 Component Basics ... 71
3.1.3 File Paths ... 73
3.2 Connections ... 78
3.2.1 Connection Types ... 81
3.2.2 Connection Settings ... 87
3.2.3 Connection Context Menu ... 89
3.2.4 Faulty Connections ... 90
3.2.5 Keep Connections after Deleting Components ... 92
3.3 General Procedures and Features ... 94
3.3.1 Validation ... 94
3.3.2 Code Generation ... 96
3.3.3 Text View Features ... 96
3.3.4 Text View Search ... 100
3.3.5 Mapping Settings ... 103

4 Structural Components 105

4.1 XML and XML Schema ... 106

4.1.1 XML Component Settings ... 107
4.1.2 Derived Types ... 111
4.1.3 NULL Values ... 113
4.1.4 Comments and Processing Instructions ... 115
4.1.5 CDATA Sections ... 116
4.1.6 Wildcards: xs:any/xs:anyAttribute ... 117
4.1.7 Custom Namespaces ... 120
4.1.8 Schema Manager ... 122

5 Transformation Components ... 138

5.1 Simple Input ... 139
5.1.1 Adding Simple Input Components ... 140
5.1.2 Simple Input Component Settings ... 140
5.1.3 Creating a Default Input Value ... 142
5.1.4 Example: Using File Names as Mapping Parameters ... 143
5.2 Simple Output ... 146
5.2.1 Adding Simple Output Components ... 147
5.2.2 Example: Previewing Function Output ... 147
5.3 Variables ... 150
5.3.1 Add a Variable ... 152
5.3.2 Scope and Context of Variables ... 156
5.3.3 Example: Filtering and Numbering Nodes ... 158
5.3.4 Example: Grouping and Subgrouping Records ... 159
5.4 Sort Components ... 162
5.4.1 Sorting by Multiple Keys ... 164
5.4.2 Sorting with Variables ... 165
5.5 Filters and Conditions ... 168
5.5.1 Example: Filtering Nodes ... 169
5.5.2 Example: Returning a Value Conditionally ... 171
5.6 Value-Maps ... 174
5.6.1 Example: Replacing Weekdays ... 177
5.6.2 Example: Replacing Job Titles ... 180
5.7 Group Functions ... 184
5.7.1 Example: Grouping Records by Key ... 187

6 Functions 190

6.1 Functions Basics ... 191
6.2 Manage Function Libraries ... 194
  6.2.1 Local and Global Libraries ... 196
  6.2.2 Relative Library Paths ... 196
6.3 User-Defined Functions ... 198
  6.3.1 UDF Basics ... 199
  6.3.2 UDF Parameters ... 204
  6.3.3 Recursive UDFs ... 207
  6.3.4 Look-up Implementation ... 210
6.4 Custom Functions ... 213
  6.4.1 Import Custom XSLT Functions ... 213
6.5 Regular Expressions ... 221
6.6 Function Library Reference ... 225
  6.6.1 core | aggregate functions ... 227
  6.6.2 core | conversion functions ... 234
  6.6.3 core | file path functions ... 244
  6.6.4 core | generator functions ... 249
  6.6.5 core | logical functions ... 250
  6.6.6 core | math functions ... 256
  6.6.7 core | node functions ... 262
  6.6.8 core | QName functions ... 267
  6.6.9 core | sequence functions ... 269
  6.6.10 core | string functions ... 295
  6.6.11 xpath2 | accessors ... 307
  6.6.12 xpath2 | anyURI functions ... 309
  6.6.13 xpath2 | boolean functions ... 310
  6.6.14 xpath2 | constructors ... 310
  6.6.15 xpath2 | context functions ... 311
  6.6.16 xpath2 | durations, date and time functions ... 314
  6.6.17 xpath2 | node functions ... 330
  6.6.18 xpath2 | numeric functions ... 336
  6.6.19 xpath2 | string functions ... 338

6.6.20 xpath3 | external information functions ... 349
6.6.21 xpath3 | formatting functions ... 352
6.6.22 xpath3 | math functions ... 356
6.6.23 xpath3 | URI functions ... 362
6.6.24 xslt | xpath functions ... 363
6.6.25 xslt | xslt functions ... 366

7 Advanced Mapping Scenarios 370

7.1 Chained Mappings ... 371
7.1.1 Example: Pass-Through Active ... 373
7.1.2 Example: Pass-Through Inactive ... 377
7.2 Mapping Node Names ... 380
7.2.1 Getting Access to Node Names ... 381
7.2.2 Accessing Nodes of Specific Type ... 389
7.2.3 Example: Map Element Names to Attribute Values ... 392
7.3 Mapping Rules and Strategies ... 397
7.3.1 Sequences ... 398
7.3.2 The Mapping Context ... 399
7.3.3 Priority context ... 408
7.3.4 Multiple target components ... 413
7.4 Processing Multiple Input or Output Files ... 417
7.4.1 Mapping Multiple Input Files to a Single Output File ... 419
7.4.2 Mapping Multiple Input Files to Multiple Output Files ... 421
7.4.3 Supplying File Names as Mapping Parameters ... 421
7.4.4 Previewing Multiple Output Files ... 422
7.4.5 Example: Split One XML File into Many ... 423

8 Automation with Altova Products 425

8.1 Automation with RaptorXML Server ... 426
8.2 MapForce Command Line Interface ... 427

9 Altova Global Resources 429

9.1 Global Resource Setup Part 1 ... 430

9.2 Global Resource Setup Part 2 ... 432
9.3 XML Files as Global Resources ... 435
9.4 Folders as Global Resources ... 437

10 Catalogs in MapForce 439

10.1 How Catalogs Work ... 440
10.2 Catalog Structure in MapForce ... 442
10.3 Customizing Your Catalogs ... 444
10.4 Environment Variables ... 446

11 Menu Commands 447

11.1 File ... 448
11.2 Edit ... 451
11.3 Insert ... 452
11.4 Component ... 455
11.5 Connection ... 457
11.6 Function ... 458
11.7 Output ... 459
11.8 View ... 461
11.9 Tools ... 463
11.9.1 Customize Menus ... 464
11.9.2 Customize Shortcuts ... 465
11.9.3 Options ... 467
11.10 Window ... 473
11.11 Help ... 474

12 Appendices 479

12.1 Support Notes ... 480
12.1.1 Supported Sources and Targets ... 480
12.1.2 Supported Features in Generated Code ... 480
12.2 Engine Information ... 482
12.2.1 XSLT and XQuery Engine Information ... 482
12.2.2 XSLT and XPath/XQuery Functions ... 487

12.3 Technical Data 582
12.3.1 OS and Memory Requirements 582
12.3.2 Altova Engines 582
12.3.3 Unicode Support 583
12.3.4 Internet Usage 583
12.4 License Information 584
12.4.1 Electronic Software Distribution 584
12.4.2 Software Activation and License Metering 585
12.4.3 Altova End-User License Agreement 586

Index **587**

1 Introduction

MapForce® 2023 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, Windows 11, and Windows Server 2008 R2 SP1 with Platform Update or newer. 64-bit support is available for the Enterprise and Professional editions.

Last updated: 7 October 2022

1.1 New Features

This section describes new features of each MapForce release. For more details, see the respective subsection.

1.1.1 Version 2023

Version 2023

- Support for the following themes has been added: *Classic*, *Light*, and *Dark*. For more information, see Window 473.
- Internal updates and optimizations.
- Eclipse support has been updated and now covers the following versions: 2022-09, 2022-06, 2022-03, 2021-12 (*Professional and Enterprise editions*).
- Support for ODETTE EDI messages (*Enterprise Edition*).
- Support for the XII Transformation Registry 5 Specification (*Enterprise Edition*).
- It is now possible to create database-based UDF parameters 205 and variables 151 with a tree of related tables (*Professional and Enterprise editions*).
- It is now possible to send an `application/x-www-form-urlencoded` request structure to a REST service (*Enterprise Edition*).
- Support for UN/EDIFACT D.21B and D.22A Directories (*Enterprise Edition*).
- Support for SQLite 3.39.2, MariaDB 10.9.2, and PostgreSQL 14.5 (*Professional and Enterprise editions*).
- Support for XML Schema Manager 122 that provides a centralized way to install and manage XML schemas for use across all Altova's XBRL-enabled applications.
- Support for mappable EDI delimiters (*Enterprise Edition*). The feature is currently supported for the following EDI standards: EDIFACT, X12, and NCPDP SCRIPT.

1.1.2 Version 2022

Version 2022 Release 2

- Internal updates and optimizations
- Eclipse support has been updated and now covers the following versions: 2021-12, 2021-09; 2021-06; 2021-03 (*Professional and Enterprise editions*).
- Support for Visual Studio 2022 in the MapForce Plug-in for Visual Studio and code generation (*Professional and Enterprise editions*).
- Support for .NET 6.0 in code generation (*Professional and Enterprise editions*).
- New database versions are supported: PostgreSQL 14, SQLite 3.37.2, MariaDB 10.6.5, MySQL 8.0.28, IBM DB2 11.5.7 (*Professional and Enterprise editions*).
- It is now possible to preview images in the **Project** window (*Professional and Enterprise editions*).
- It is now possible to create EBA-conformant filing indicators for target XBRL components (*Enterprise Edition*).

Version 2022

- Internal updates and optimizations
- Eclipse support has been updated and now covers the following versions: 2021-09; 2021-06; 2021-03; 2020-12 (*Professional and Enterprise editions*).
- Copy-all connections 86 now support JSON. This feature is available only for compatible JSON types (*Enterprise Edition*).
- A new StyleVision output pane called *Text* has been introduced. If an SPS file is attached to a component, the new plain text output format can be previewed in MapForce (*Professional and Enterprise editions*).
- Support for JSON Schema in variables 150 and UDF parameters 204 (*Enterprise Edition*).
- Support for NoSQL databases: MongoDB and CouchDB (*Enterprise Edition*).
- A new `bson` function library has now become available, which allows you to create and manipulate some of the BSON types (*Enterprise Edition*).
- Support for UN/EDIFACT D.20B and D.21A Directories.
- Support for SWIFT 2021.

1.1.3 Version 2021

Version 2021 Release 3

- Support for new JSON Schema Draft 2019-09 and Draft 2020-12 (*Enterprise Edition only*).

Version 2021 Release 2

- XSLT 3.0 is now supported as mapping language. See Generating XSLT Code 96. MapForce now includes new built-in functions that are supported when the mapping language is XSLT 3.0. For more information, see Function Library Reference 225.
- Internal updates and optimizations.

Version 2021

- Internal updates and optimizations.

1.1.4 Version 2020

Version 2020 Release 2

- A new Manage Libraries window 24 is available that enables you to view and manage all function libraries imported at document and at program level (this includes MapForce user-defined functions and other kinds of libraries). This makes it possible, for example, to easily copy-paste user-defined functions from one mapping to another, see Copy-Pasting UDFs Between Mappings 203.

- When a mapping file imports libraries, the path of imported library files is relative to the mapping file by default, see Relative Library Paths 196. You can still import mappings at application level, like in previous releases, but in this case the library path is always absolute.
- If a mapping file imports XSLT libraries, you can generate XSLT code that references the imported library files using a relative path. The new option is available in the Mapping Settings 103 dialog box.
- Internal updates and optimizations

Version 2020

- When replacing values with the help of a look-up table, you can paste tabular data (key-value pairs) from external sources such as CSV or Excel into the mapping. Also, it is easier to handle cases when a value is not found in the predefined look-up table—processing such values no longer requires the use of **`substitute-missing`** function. See Using Value-Maps 174.
- Internal updates and optimizations

1.1.5 Version 2019

Version 2019 Release 3

- Major parts of the graphical user interface are now optimized for monitors with high pixel density (HiDPI)
- Support for explicitly setting the Java Virtual Machine path from MapForce, see Java Settings 469
- Internal updates and optimizations

Version 2019

- Internal updates and optimizations

1.2 What Is MapForce?

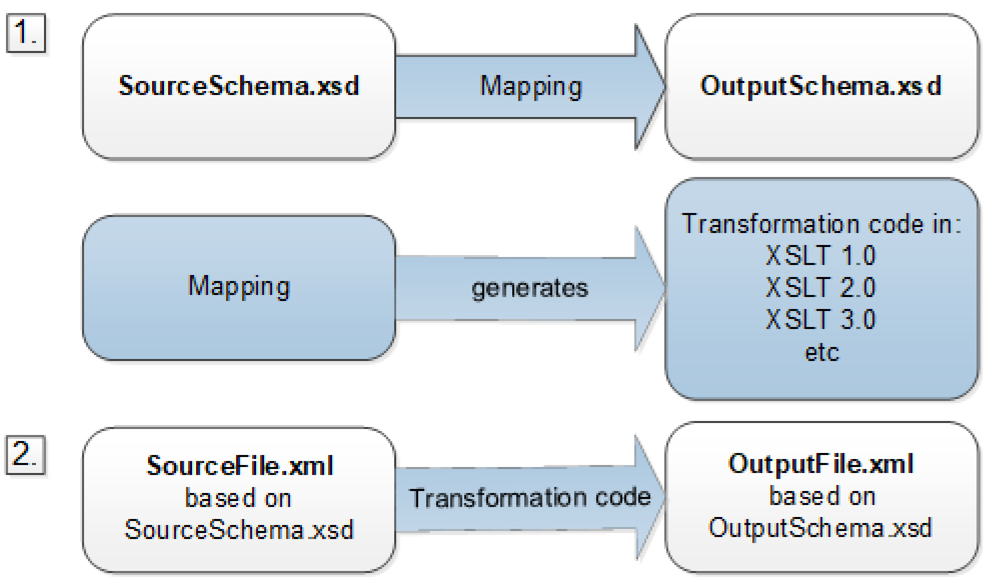
Altova website: Data mapping tool

MapForce is a powerful and flexible tool that allows any-to-any graphical mapping of different data formats. See Mapping: Sources and Targets (15) for a complete list of available data formats. MapForce enables you to map one source to one target, one source to multiple targets, multiple sources to one target, or multiple sources to multiple targets. To find out more about mapping scenarios, see Mapping Scenarios (17) and Tutorials (31). MapForce also provides an extensive range of data processing and filtering options, such as functions (190), variables (150), filters and conditions (168), that allow you to manipulate your data.

In order to be able to carry out a mapping, you must provide a data structure that describes the structure of each of your source and target files. For example, an XML schema defines the structure of an XML document. The mapping (from source to target) is achieved by means of a drag-and-drop graphical user interface. You do not have to write any program code for the mapping. The code is generated for you by MapForce. You can then use this code to transform documents having the source data structure to documents having the target data structure.

Abstract model

The abstract model below illustrates one of the basic scenarios of data transformation in MapForce. The first step shows that one abstract structure called `SourceSchema.xsd` is mapped to another abstract structure called `OutputSchema.xsd`. The mapping generates transformation code in the selected transformation language. The second step shows that the content of the source XML file, which is based on `SourceSchema.xsd`, is mapped to the target XML file, which is based on `OutputSchema.xsd`. The mapping of the content from the source to the target file is carried out by means of the transformation code generated in the previous step.



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

Conventions

Mapping files illustrated or referenced in the manual can be found in the following locations:

- `C:\Users\<username>\Documents\Altova\MapForce2023\MapForceExamples`
- `C:\Users\<username>\Documents\Altova\MapForce2023\MapForceExamples\Tutorial`
- `C:\Users\<username>\Documents\Altova\MapForce2023\MapForceExamples\Tutorial\BasicTutorials`

In this section

This section is organized into the following topics:

- Mapping: Sources and Targets 15
- Transformation Languages 16
- Mapping Scenarios 17
- Integration with Altova Products 19

1.2.1 Mapping: Sources and Targets

In MapForce, *source* and *target* are essential terms that refer to data structures from which or to which data is mapped, respectively. Technologies that can be used as mapping sources and targets are listed below.

MapForce Basic Edition

- XML and XML Schema

MapForce Professional Edition

- XML and XML Schema
- Flat files, including comma-separated values (CSV) and fixed-length field (FLF) format
- Databases: all major relational databases, including Microsoft Access and SQLite databases
- Binary files (raw BLOB content)

MapForce Enterprise Edition

- XML and XML Schema
- Flat files, including comma-separated values (CSV) and fixed-length field (FLF) format
- Data from legacy text files can be mapped and converted to other formats with MapForce FlexText
- SQL Databases: all major relational databases, including Microsoft Access and SQLite databases
- NoSQL Databases
- Binary files (raw BLOB content)
- EDI family of formats, including UN/EDIFACT, ANSI X12, HL7, IATA PADIS, SAP IDoc, TRADACOMS
- JSON files
- Microsoft Excel 2007 and later files
- XBRL instance files and taxonomies
- Protocol Buffers

1.2.2 Transformation Languages

In MapForce, a transformation language is used to generate transformation code that carries out mappings. You can select/modify a transformation language at any time. MapForce allows viewing the transformation code in the selected language. For more information, see Code Generation 96. You can also generate this code via the menu command **File | Generate Code in** and use this code for transforming any data document that is valid according to the source component's schema. Depending on the MapForce edition, you can choose the following languages for your data transformation:

MapForce Basic Edition	MapForce Professional and Enterprise editions
• XSLT 1.0 • XSLT 2.0 • XSLT 3.0	• XSLT 1.0 • XSLT 2.0 • XSLT 3.0 • BUILT-IN • XQuery • Java • C# • C++

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

- In 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 (*shown below*).

When you change the transformation language of the mapping, certain MapForce features may not be supported for that language. For more information, see Support Notes 480.

As you design or preview mappings, MapForce validates the integrity of your schemas or transformations. If any validation errors occur, MapForce displays them in the Messages window 26. This is helpful because you can immediately review and correct these errors.

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

BUILT-IN
When you select BUILT-IN as a transformation language for your mapping, MapForce uses its native transformation engine to execute the data mapping. MapForce also uses this option implicitly whenever you preview the output of a mapping where the selected transformation language is Java, C#, or C++.

The BUILT-IN engine executes mappings without the need for any external processors, which may be a good choice if memory usage is an issue. If you do not need to generate program code in a specific language, use BUILT-IN as a default option, because it supports most MapForce features compared to other languages (see

Support Notes (480)). Furthermore, if you select BUILT-IN as a transformation language, you will be able to automate the mapping with MapForce Server. For more information, see Automation with Altova Products (425).

1.2.3 Mapping Scenarios

The scenarios can differ on the following criteria: (i) sources and targets, and (ii) complexity of mappings. Different data structures can be used as sources and targets: e.g., XML Schema, an XML file with an assigned schema, databases etc. To find out more about the acceptable formats of sources and targets, see Mapping: Sources and Targets (15).

The complexity of mapping designs is illustrated in but not limited to the following scenarios:

- Mapping one source to one target. For more information about this type of mapping, see Tutorial 1 (32).
- Merging multiple data sources into one target. For more information, see Tutorial 2 (42).
- Filtering the data in such a way that only a subset of this data is mapped to the target file. See Tutorial 3 (49).
- Mapping the structure and content of the source to the target file. See Tutorial 4 (56).

Regardless of the technology you work with, MapForce typically determines automatically the structure of your data or suggests supplying a schema for your data. MapForce can also generate schemas 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, MapForce makes the data inside the XML file available for mapping to other files or formats. To find out more about the basic terms and features of MapForce, see Basic Tasks (65) and User Interface Overview (21).

For easier access and management, you can organize your data mapping designs into mapping projects. This feature is available for MapForce Professional and Enterprise editions. In addition to generating code for individual mappings within the project, you can generate program code from entire projects.

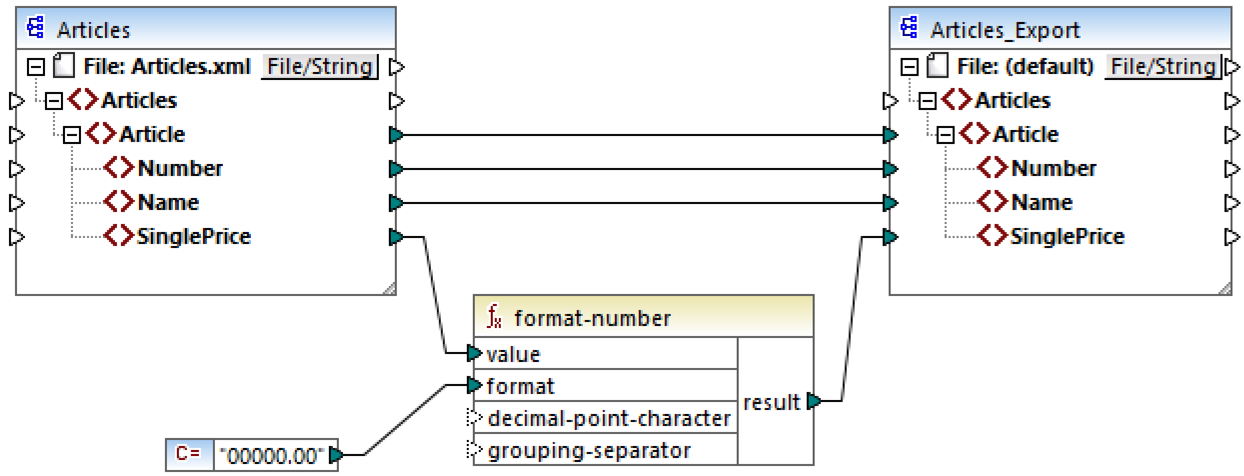
In MapForce, you can completely customize not only the look and feel of the development environment, but also various settings related to each technology and mapping component type. For example:

- When mapping to or from XML, you can choose (i) whether to include a schema reference, or (ii) 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. It is also possible to choose whether MapForce should use database transactions, or whether it should strip the database schema name from table names when it generates code.
- In the case of XBRL, you can select the structure views that MapForce should display: the **Presentation and definition linkbases** view, the **Table Linkbase** view, or the **All concepts** view.

The examples below illustrate mapping designs that use the same (*Example 1*) and different (*Example 2*) types of source and target structures. Both mapping examples are simple in that only one data source and one data target are used. To find out about more advanced mappings, tasks and procedures, see Advanced Mapping Scenarios (370).

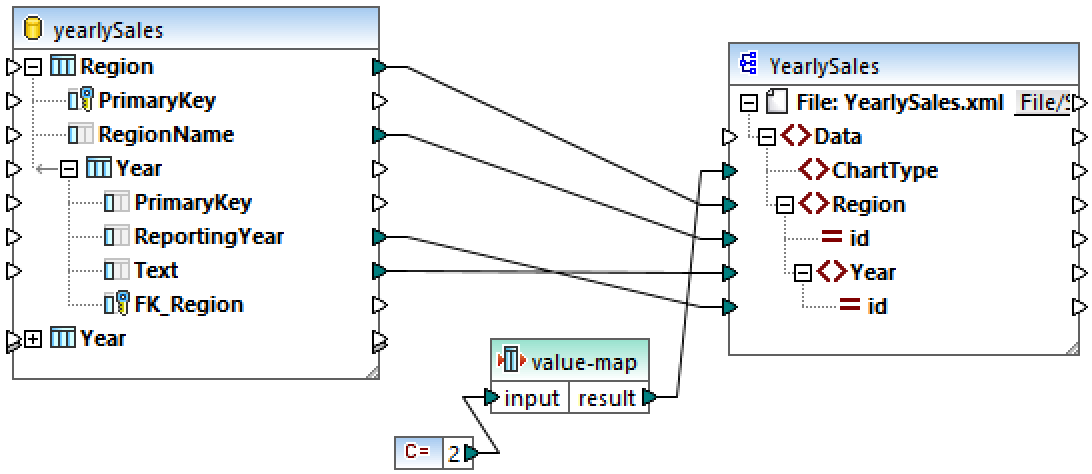
Example 1: XML mapping

MapForce allows designing all mapping transformations visually. For example, in the case of XML, you can connect any element and attribute in an XML file to an element or attribute of another XML file. Thus, you instruct MapForce to read the data from the source node and write it to the target node. The transformation of one XML file into another XML file is illustrated below:



Example 2: Database mapping

When you work 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 you set up a database connection from MapForce, you can flexibly choose a database driver and a connection type (ADO, ADO.NET, ODBC, or JDBC) depending on your existing infrastructure and data mapping needs. Additionally, you can visually build SQL queries, use stored procedures or query a database directly (support depends on the database type, edition and driver). An example of data transformation from a database into an XML file is given below:



1.2.4 Integration with Altova Products

Transformations can be run inside MapForce using built-in XSLT/XQuery engines. MapForce can also be used in tandem with other Altova products (*see below*).

XMLSpy
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 menu command **Component | Edit Schema Definition in XMLSpy** is available when you click an XML component.
RaptorXML Server
You can choose to run the generated XSLT code 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.

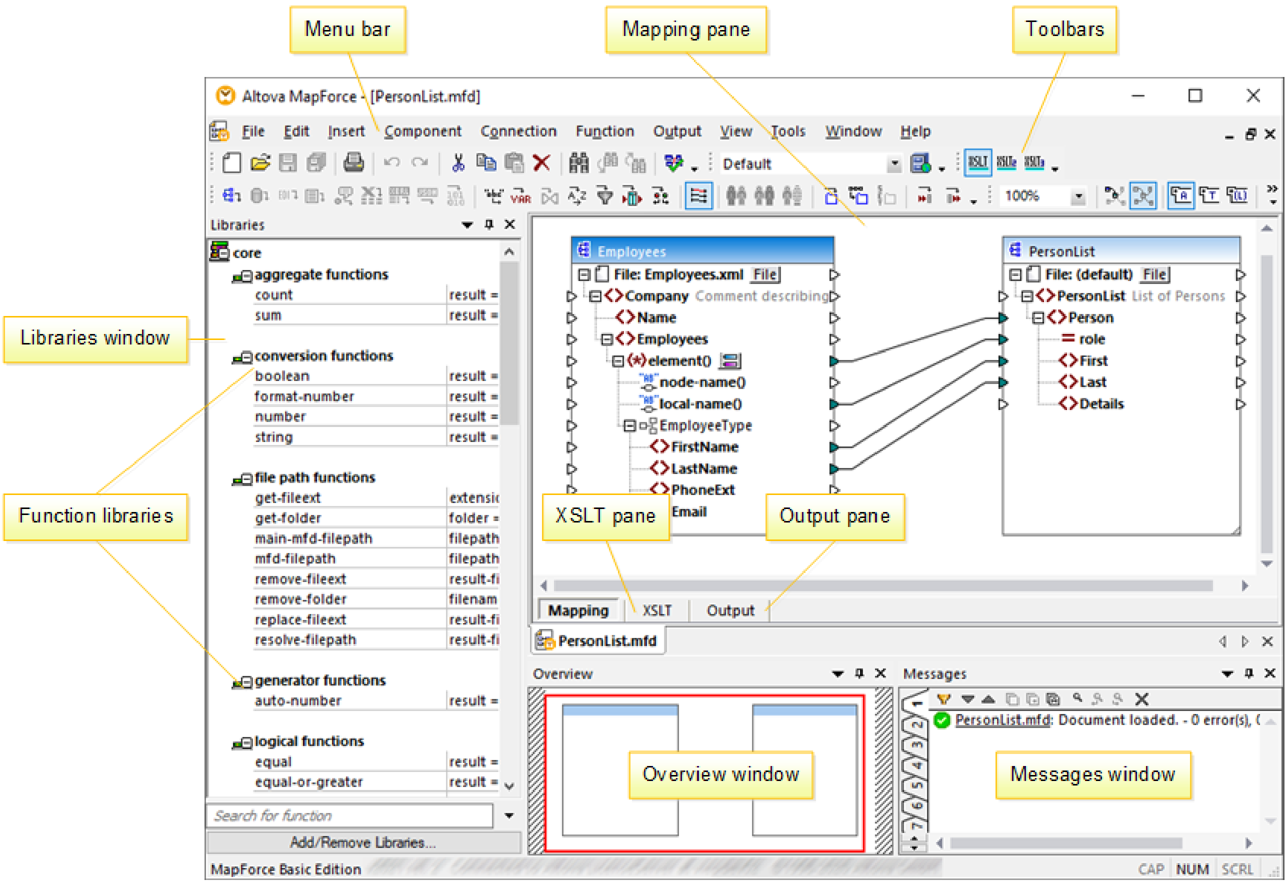
MapForce Server (Enterprise and Professional editions)
You can automate MapForce tasks with the help of Altova MapForce Server, which can be installed on Windows, Linux, and macOS systems. MapForce Server enables you to run the transformations specified in a mapping, not only from the command line of the respective OS but also through API calls (.NET, COM, Java).
FlowForce Server (Enterprise and Professional editions)
You can also automate MapForce tasks with the help of Altova FlowForce Server, which can be installed on Windows, Linux, and macOS systems. FlowForce Server enables you to carry out MapForce Server tasks according to a schedule.
StyleVision (Enterprise and Professional editions)
With the help of StyleVision, 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 as a plug-in
MapForce Professional and Enterprise editions 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 the MapForce functionality without leaving your preferred development environment.

For more information about automating tasks, see Automating MapForce Tasks with Altova Products[425].

1.3 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**.

The image below illustrates the main parts of the MapForce graphical user interface.



For more information about the features and functions of each part, see the respective topic below.

In this section

This section is organized in the following way:

- Bars [22]
- Windows [22]
- Messages Window [26]
- Panes [27]

1.3.1 Bars

This topic gives an overview of the available bars.

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 a desired location. The screenshot below illustrates the the **Menu** bar and toolbars. The actual interface depends on your MapForce edition and the settings you choose.

Application status bar

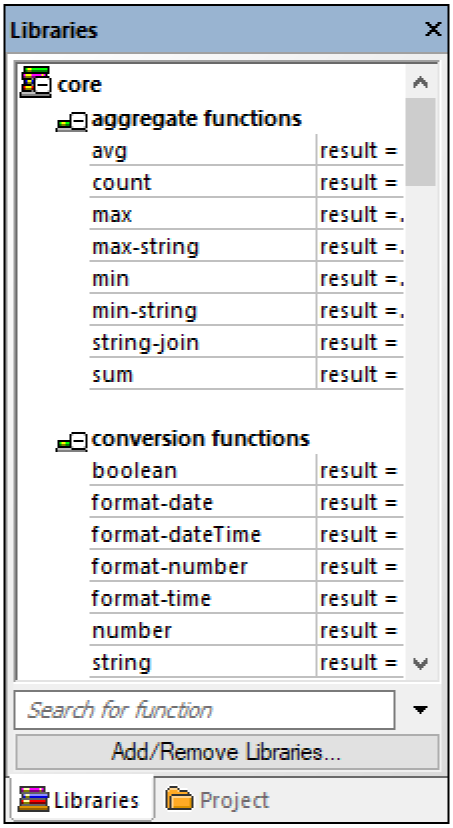
The application status bar appears at the bottom of the MapForce window and shows application-level information. Tooltips are displayed when you move the mouse over a toolbar button. If you are using the 64-bit version of MapForce, the application name appears in the status bar with the x64 suffix. There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

1.3.2 Windows

This topic gives an overview of the available windows.

Libraries window

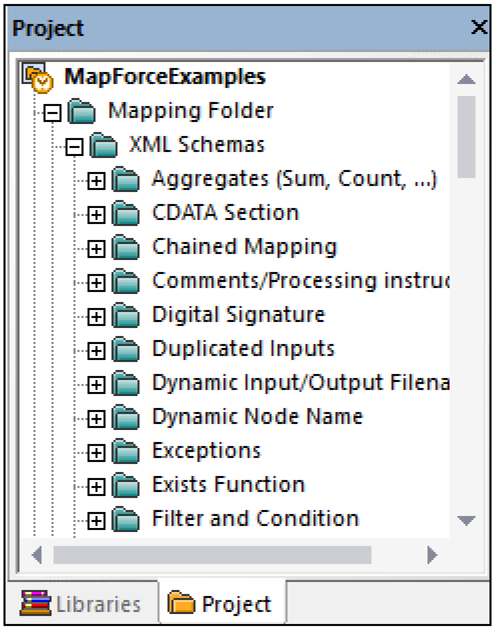
The **Libraries** window lists the MapForce built-in functions organized by library. The list of available functions changes depending on the transformation language you select either from the **Output** menu or from the **Language Selection** toolbar. For more information, see Transformation Languages [16]. If you have created user-defined functions or 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 Functions 190.

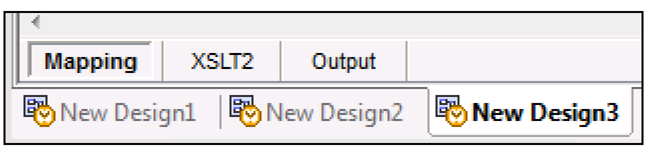
Project window (Enterprise and Professional editions)

MapForce supports the Multiple Document Interface and allows grouping your mappings into mapping projects. The **Project** window shows all files and folders that have been added to the project. Project files have a **`*.mfp`** (MapForce Project) extension. To search for mappings inside projects, click anywhere inside the **Project** window and press **CTRL + F**.



Mapping window(s)

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 under the **Mapping** pane (*see screenshot below*).



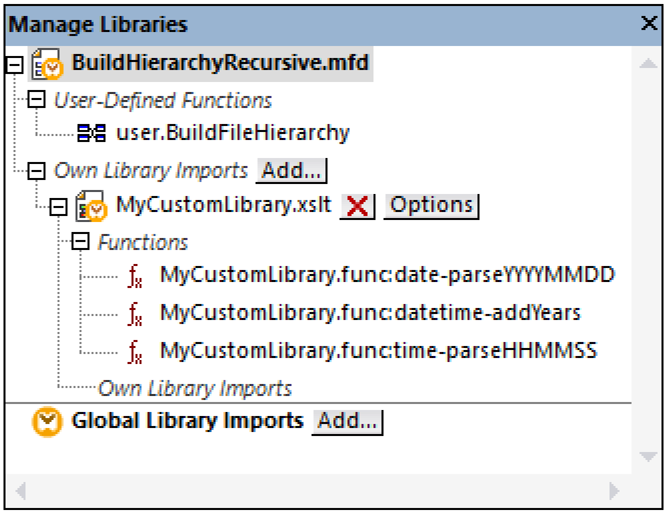
You can access Window management options using the menu command **Window | Windows**. The **Windows** dialog box allows you to perform various actions including activating, saving, closing, or minimizing open mapping windows. To select multiple windows in the **Windows** dialog box, click the required entries while holding the **Ctrl** key pressed.

Manage Libraries window

From this window you can view and manage all user-defined functions (UDFs) and imported custom libraries that are used by the currently open mappings.

By default, the **Manage Libraries** window is not visible. To display it, do one of the following:

- In the **View** menu, click **Manage Libraries**.
- Click **Add/Remove Libraries** at the bottom of the **Libraries** window.



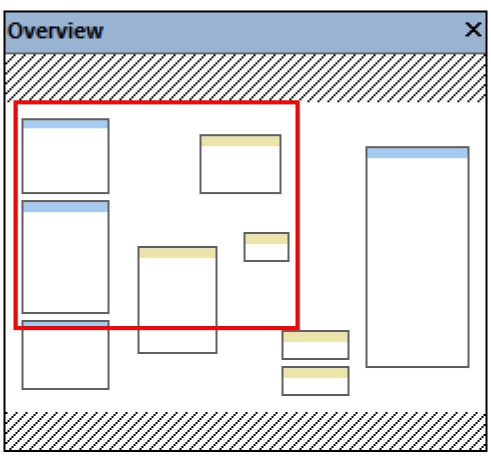
You can choose to view UDFs and libraries only for the mapping document that is currently active or for all open mapping documents. To view imported functions and libraries for all of the currently open mapping documents, right-click inside the window and select **Show Open Documents** from the context menu.

To display the path of the open mapping document instead of the name, right-click inside the window and select **Show File Paths** from the context menu.

For more information, see Manage Function Libraries 194.

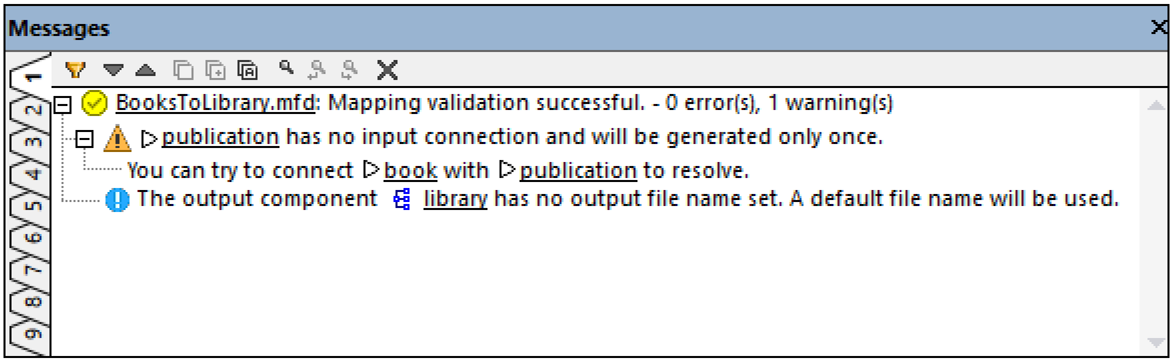
Overview window

The **Overview** window gives a bird's-eye view of the Mapping pane 27. Use it to navigate quickly to a particular location in the mapping area when the size of the mapping is very large. To navigate to a particular location in the mapping, click and drag the red rectangle.



1.3.3 Messages Window

The **Messages** window (*see screenshot below*) shows validation statuses, messages, errors, and/or warnings when you preview or validate 94 a mapping. Click the underlined text in the **Messages** window to see a component or structure which caused the information, warning, or error message.



Validation status icons

When you validate a mapping, MapForce checks, for example, for unsupported component kinds, incorrect or missing connections. The validation result is 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 **Messages** window may additionally display any of the following message types: information messages, warnings, and errors.

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

To highlight the component or structure which caused the information, warning, or error message, click the underlined text in the **Messages** window.

Message-related actions

The **Messages** window enables you to take the following actions:

Icon	Description
	Filter messages by severity: information messages, errors, and warnings. Select **Check All** to include all severity levels (this is the default behavior). Select **Uncheck All** to remove all severity levels from the filter. In this case, only the general execution or validation status message is displayed.
	Jump to the next line.
	Jump to the previous line.
	Copy the selected line to the clipboard.
	Copy the selected line to the clipboard, including any lines nested under it.
	Copy the full contents of the **Messages** window to the clipboard.
	Find a specific text in the **Messages** window. Optionally, to find only words, select **Match whole word only**. To find text while preserving the upper or lower case, select **Match case**.
	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 validating the mapping.

1.3.4 Panes

This topic gives an overview of the available panes.

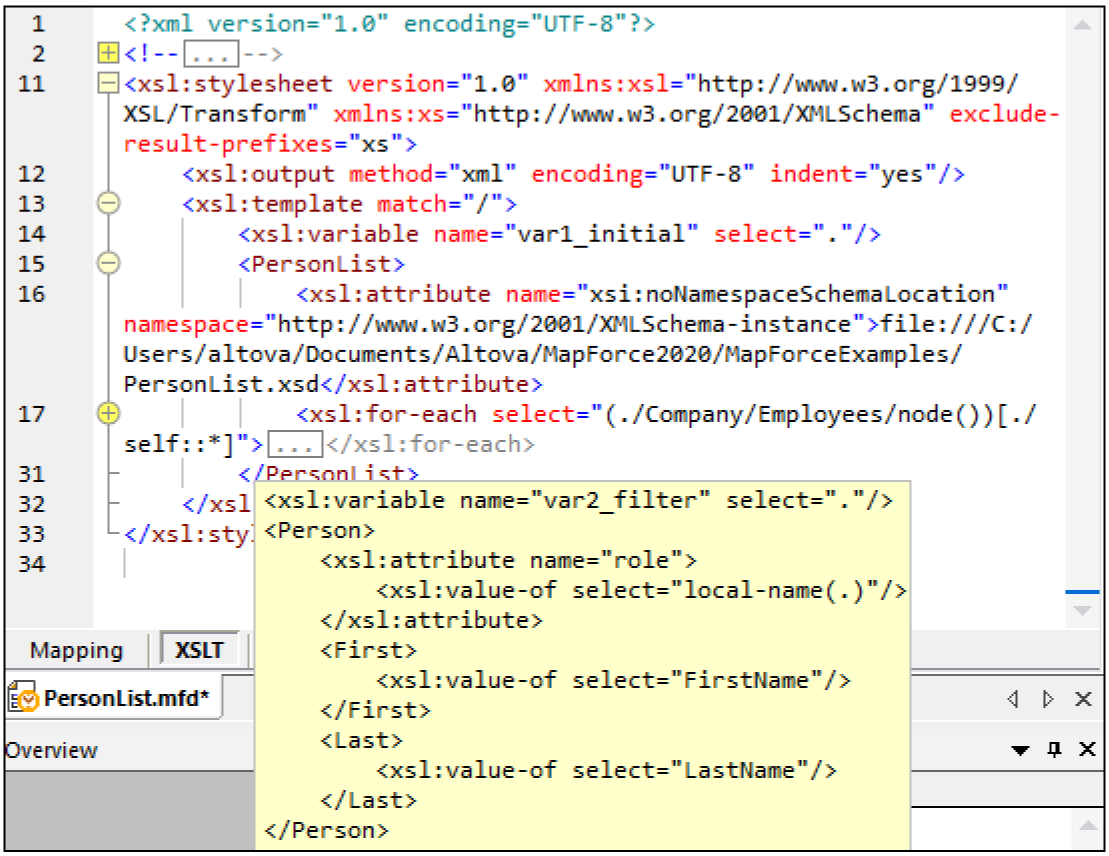
Mapping pane

The **Mapping** pane is the working area where you design mappings[94]. You can add mapping components (e.g., files, schemas, constants, variables, and so on) to the mapping area from the **Insert** menu. For more information, see Add Components to Mapping[69]. You can also drag functions from the **Libraries** window into the **Mapping** pane. For details, see Add a Function to the Mapping[191].

XSLT pane

The **XSLT** pane displays the XSLT transformation code generated from your mapping. To switch to this pane, select XSLT, XSLT 2 or XSLT3 as a transformation language [16] and click the tab with the same name.

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 into the tooltip, an additional ellipsis appears at the end of the tooltip.



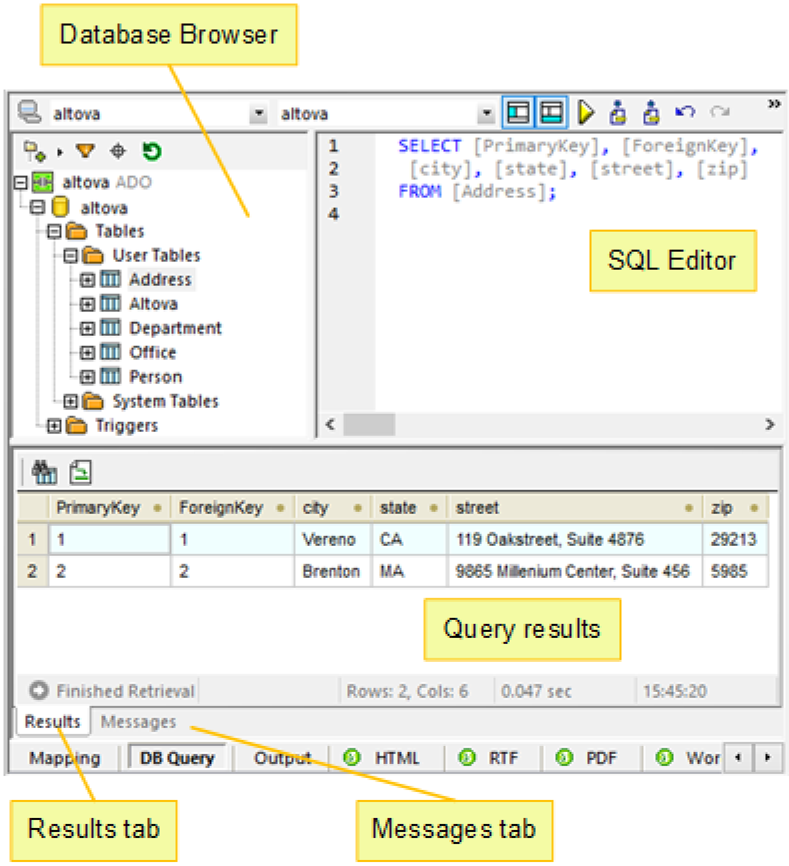
To configure the display settings, including the indentation, end of line markers, and others, right-click the pane and select **Text View Settings** from the context menu. Alternatively, click (**Text View Settings**) in the toolbar.

XQuery pane (Enterprise and Professional editions)

The **XQuery** pane displays the XQuery transformation code generated from your mapping when you click the **XQuery** button. This pane is available when you select XQuery as a transformation language. This pane also provides line numbering and code folding functionality, which works in a similar way as in the XSLT pane (see above).

DB Query Pane (Enterprise and Professional editions)

The **DB Query** pane allows you to directly query any major database. You can work with multiple active connections to different databases.



Output pane

The **Output** pane displays the result of the mapping transformation. 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).

StyleVision Output Panes (Enterprise and Professional editions)

If you have installed Altova StyleVision, the StyleVision output panes will become available next to the **Output** pane. The StyleVision output panes enable you to preview and save the mapping output in HTML, RTF, PDF, and Word 2007+ formats. This is possible thanks to StyleVision Power Stylesheet (SPS) files designed in StyleVision and assigned to a mapping component in MapForce.

2 Tutorials

With the help of these tutorials, you will be able to understand and use the basic data transformation capabilities of MapForce. You will be guided through the basics step by step. The tutorials gradually grow in complexity. Therefore, it is recommended to follow them sequentially. Basic knowledge of XML and XML Schema will be advantageous.

Example files
The mapping files illustrated or referenced in these tutorials are available in the BasicTutorials folder 15. When you are in doubt about the possible effects of changing the original MapForce examples, create back-ups before changing them.

List of tutorials

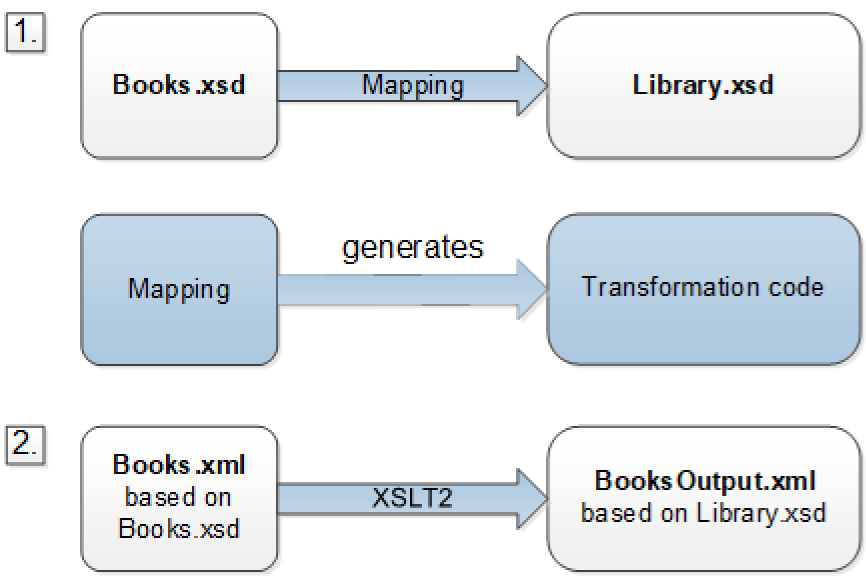
One source to one target
This tutorial 32 shows how to use key MapForce mechanisms to map the nodes of a source file to the nodes of a target file. The tutorial goes on to explain how to convert an XML file defined by one XML schema to an XML file defined by a different XML schema.
Multiple sources to one target
This tutorial 42 shows how to merge data from multiple source XML files to one target file.
Chained mappings
In this tutorial 49, we create a simple mapping as in the first tutorial, then filter the data produced by this mapping and pass the filtered data to the second target file.
Multiple sources to multiple targets
This tutorial 56 shows how to read data from multiple XML instance files located in the same folder and write this data to multiple XML files generated on the fly.

2.1 One Source to One Target

This tutorial describes how to create a mapping for one of the most basic scenarios. Our goal is to take the data from XML file A with XML schema A assigned to it and put this data into XML file B with XML schema B assigned to it. Thus, our target file will have the same data as in the source but with a different schema (structure). Thanks to the transformation code, you will be able to see how the structure of the source file has changed. Note the mapping could be carried out only between the structures, but we will not be able to preview the result of the transformation in the Output pane [29]. Therefore, for illustration purposes, we use an XML file called **`Books.xml`** (*see code listing below*). The broad outline of our method will be as follows:

1. Since we are using two data structures, we will create two components (*Source* and *Target*) in our mapping design.
2. Then we need to map nodes by connecting a source node to the desired target node. It is these connections that constitute the mapping and determine what source node maps to what target node.
3. Since the transformation of one document into another is carried out by using a suitable transformation language, such as XSLT, we select a transformation language.
4. We use MapForce's built-in transformation engines to transform the source XML document (**`Books.xml`**) into the required target document. This target document will be an XML document that is valid according to the target XSD (**`Library.xsd`**).
5. Finally, we can save the output XML file.

The image below illustrates an abstract model of the data transformation used in this tutorial:



The abstract model above shows two steps of the mapping process. In the first step, the structure of **`Books.xsd`** is mapped to a new structure called **`Library.xsd`**. The mapping is carried out by means of a transformation language [16]. The choice of a transformation language depends on your MapForce edition. In our case, XSLT2 is chosen as a transformation language. In the second step, the content of **`Books.xml`**, which has **`Books.xsd`** assigned to it, is mapped to the target file (**`BooksOutput.xml`**) and based on a new schema (**`Library.xsd`**). The code listing below shows sample data from **`Books.xml`** that will be used as a data source.

```
<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>
```

This is how we want our data to look in the target file called **BooksOutput.xml**:

```
<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>
```

Some element names in the source and target XML are not the same. Our goal is to populate the elements <author>, <title>, <genre> and <publish_year> of the target file with the content of the equivalent elements in the source file (<author>, <title>, <category>, <year>). The attribute id in the source file must be mapped to the <id> element in the target file. Finally, we must populate the <last_updated> element of the target file with the date and time indicating when the file was last updated.

To carry out the required data transformation, take the steps described in the subsections below.

2.1.1 Create and Save Design

This topic explains how to create a new design, select a transformation language, validate and save your mapping.

Create a new design

To be able to carry out a transformation, you will need to create a new mapping design, which can be done in one of the following ways:

- Go to the **File** menu and click **New**.

- Click in the toolbar.

Select a transformation language

Depending on your MapForce edition, different transformation languages 16 are available. For this tutorial, we have selected XSLT2. You can select this transformation language in one of the following ways:

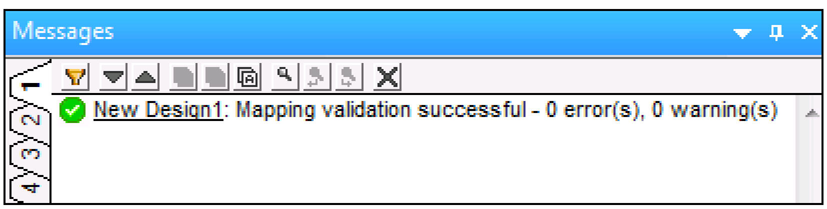
- Click (**XSLT2**) in the toolbar.
- In the **Output** menu, click **XSLT 2.0**.

Validate and save the design

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

- In the **File** menu, click **Validate Mapping**.
- Click (**Validate**) in the toolbar.

The **Messages** window displays the validation results as follows:



To save the mapping, do one of the following:

- Click **Save** in the **File** menu.
- Click (**Save**) in the toolbar.

For your convenience, the mapping created in this tutorial is saved as **`Tut1_SchemaToSchema.mfd`**.

2.1.2 Add Source Component

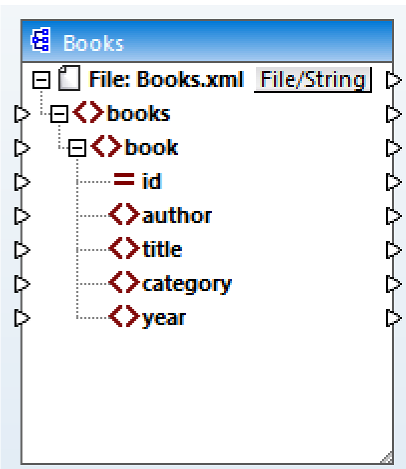
At this stage, we want to add an XSD file that will be the structure of the first component and an XML file that will provide the data for this component. The source file called **`Books.xsd`** can be added to the mapping in one of the following ways:

- Click (**Insert XML Schema/File**) in the toolbar.
- In the **Insert** menu, click **XML Schema/File**.
- Drag **`Books.xsd`** from the Windows Explorer into the mapping area.

When you add a schema, MapForce suggests adding a sample XML file. Click **Browse** and search for **`Books.xml`** that is located in the same folder. Thus, our source file contains both a schema and content. In the properties of every source or target component, we can specify an XSD file and XML fie. The XSD file defines the structure of the document in that component. The XML file provides the data of that component (source or target) and must be valid by the schema of that component. If a component is created from an XSD file, then you are prompted for an XML file that will be used as the component's data file. If a component is created from an XML file, then the XSD that is referenced from the XML file will be used to define the structure of the component's data. If no reference to an XSD exists, MapForce will ask you if it may generate an XSD file for this component.

View the structure

Now that the source file has been added to the mapping area, you can see its structure. In MapForce, this structure is known as a mapping component or simply a component[65]. You can expand elements in the component by clicking the ⊞ icon. Alternatively, you can press the **+** key on the numeric keypad. The screnshot below illustrates the source component:



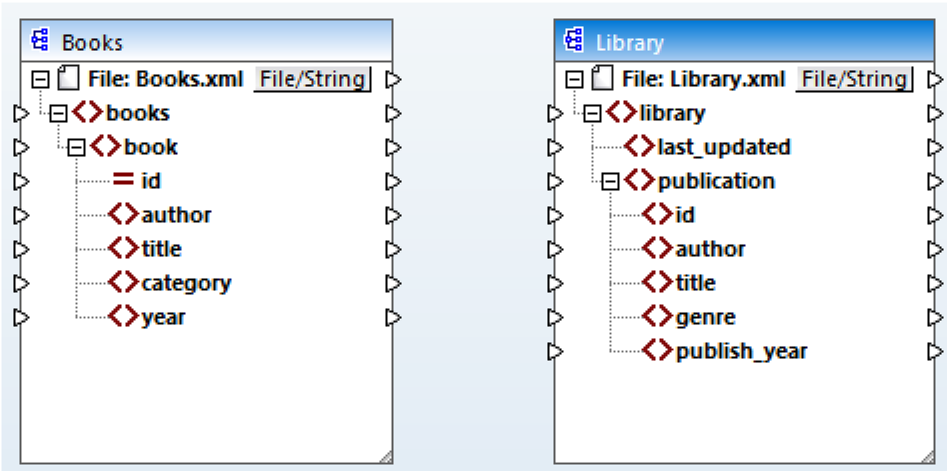
`Books` in the title bar indicates the name of the component. The top level node represents the name of the XML instance file: **`File: Books.xml`**. The XML elements in the structure are represented by the icon. XML attributes are represented by the icon. The small triangles, displayed on both sides of the component, represent data inputs on the left side and outputs on the right side. In MapForce, these triangles are called *input connectors* and *output connectors*, respectively.

Move and resize components

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 bottom right-hand corner of the component. You can also double-click this corner so that MapForce adjusts the size automatically.

2.1.3 Add Target Component

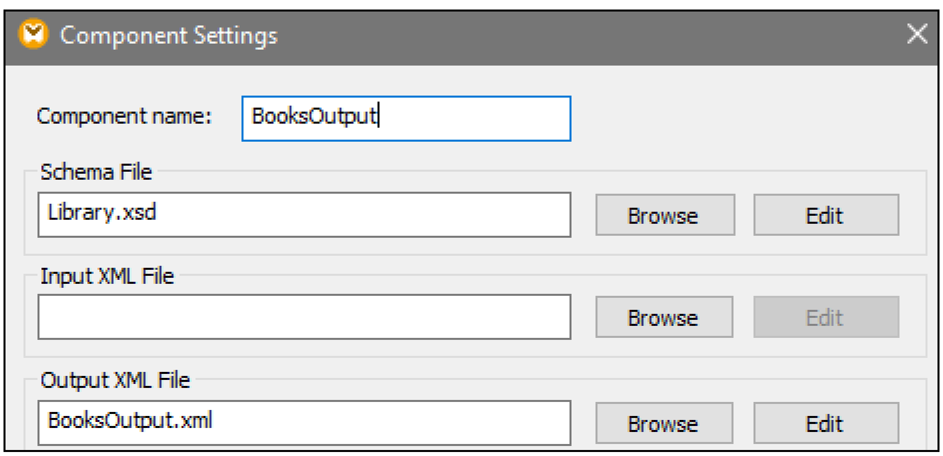
The next step is to add a target component and define its settings. To add the target file called **`Library.xsd`** to the mapping, click (**Insert XML Schema/File**). Click **Skip** when MapForce suggests supplying an instance file. At this stage, the mapping design looks as follows:



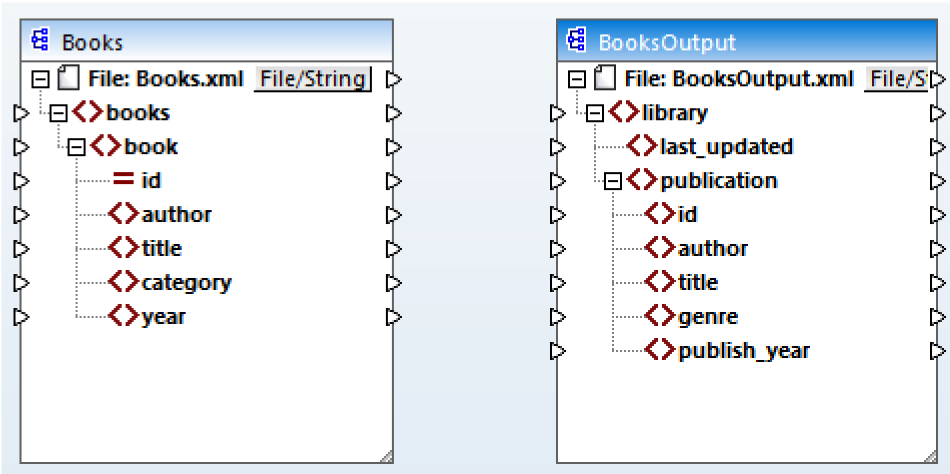
Note that when you open **Library.xsd**, it is displayed as an XML file in the component. In fact, MapForce only creates a reference to the XML file called **Library.xml**, but this XML file itself does not yet exist. Thus, out target component has a schema but no content.

Component settings

Now we need to rename the target component **BooksOutput.xml**. This will allow us to avoid confusion in the next tutorials, as we are going to use a separate file called **Library.xml**, which has its own content and is based on the same **Library.xsd** schema. In order to rename the target file, double-click the header of the target component. This opens the Component Settings dialog box 107 (*see screenshot below*), in which we need to change the name of the target file as follows:



The mapping design now looks as follows:



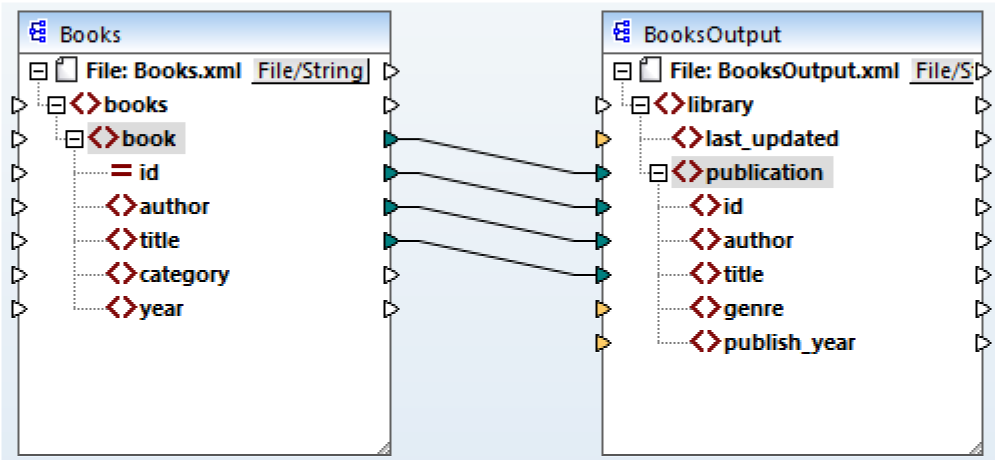
Note: The name of the header only refers to the name of the component and not the name of the schema this file is based on. To see the name of the schema, open the **Component Settings** dialog box.

2.1.4 Connect Source and Target

In this step, we will map the data in the source file to the target file. The goal is to map two types of input to the target nodes: (i) source nodes and (ii) data, both of which are mapped simultaneously. Most of the data comes from **`Books.xml`**. In our example, we will also supply information about the current date and time using the XPath function `current-dateTime` [312].

Automatic connections

We will now create a mapping connection between the `<book>` element in the source component and the `<publication>` element in the target component. To do this, 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 the child elements of `<book>` in the source file to the elements with the same names in the target file. In our example, four connections have been created simultaneously (*see screenshot below*). This feature is called *Auto Connect Matching Children* and can be disabled and customized if necessary.



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

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

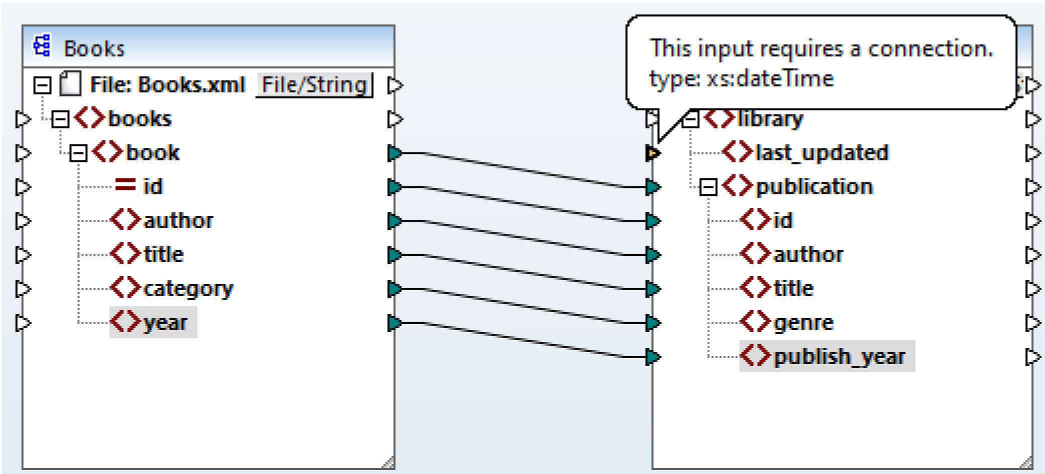
Connect mandatory items

Notice that some of the input connectors in the target component have been highlighted by MapForce in orange, which indicates that these items are mandatory. They are mandatory, because they were set in such a way in the the file's schema. To ensure the validity of the target XML file, provide values for the mandatory items as follows:

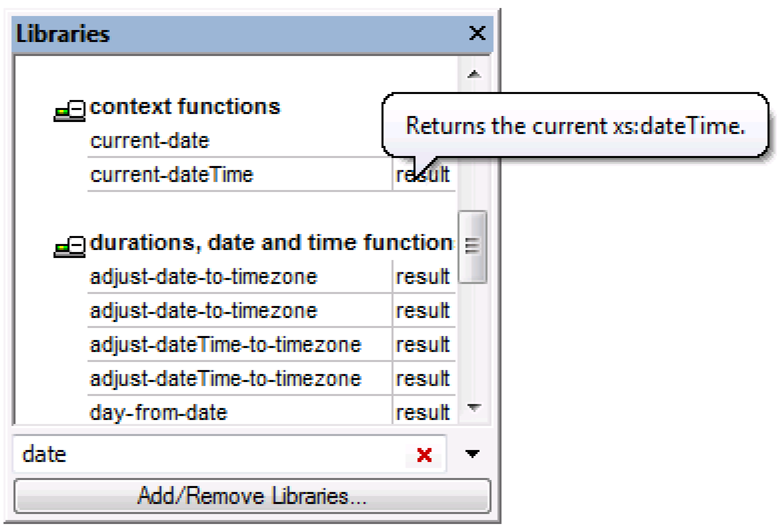
- Connect the `<category>` element in the source with the `<genre>` element in the target component.
- Connect the `<year>` element in the source with the `<publish_year>` element in the target component.

Add the current date and time

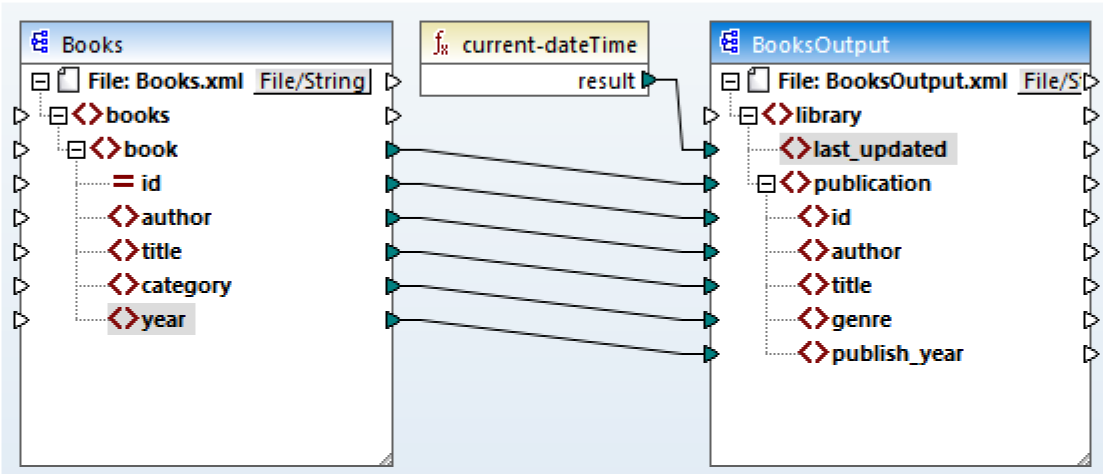
Finally, you need to supply a value for the `<last_updated>` element. If you hover over its input connector, you can see that the element is of type `xs:dateTime` (*see screenshot below*). To be able to see tips, press the toolbar button (**Show tips**). By clicking (**Show Data Types**) in the toolbar, you can also make the data type of each item visible at all times.



You can get the current date and time by means of an XSLT2 function **current-dateTime**. To find this function, type it in the text box located at the bottom of the Libraries window[22] (*see screenshot below*). Alternatively, double-click an empty area inside the **Mapping** pane and start typing `current-date`.



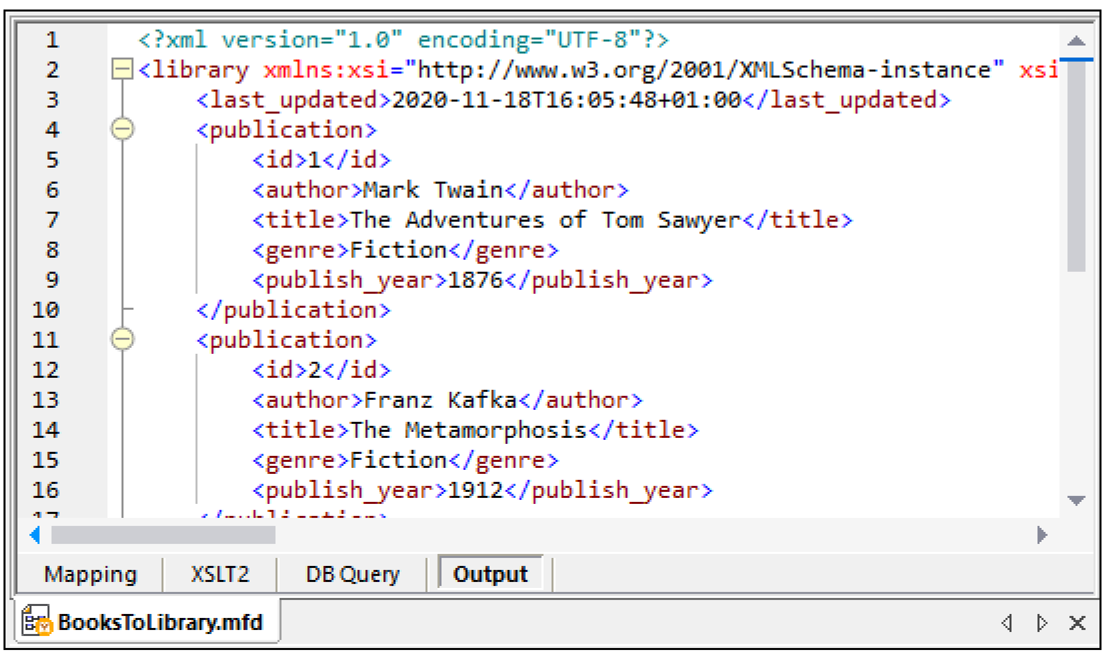
To add the function to the mapping, drag the function into the **Mapping** pane and connect its output to the input of the <last_updated> element (*see screenshot below*).



You can now validate and save your mapping, as shown in Create and Save Design[33].

2.1.5 Preview Mapping Result

MapForce uses its built-in engines to generate the output and allows previewing the result of the mapping directly in the **Output** pane (*see screenshot below*).



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

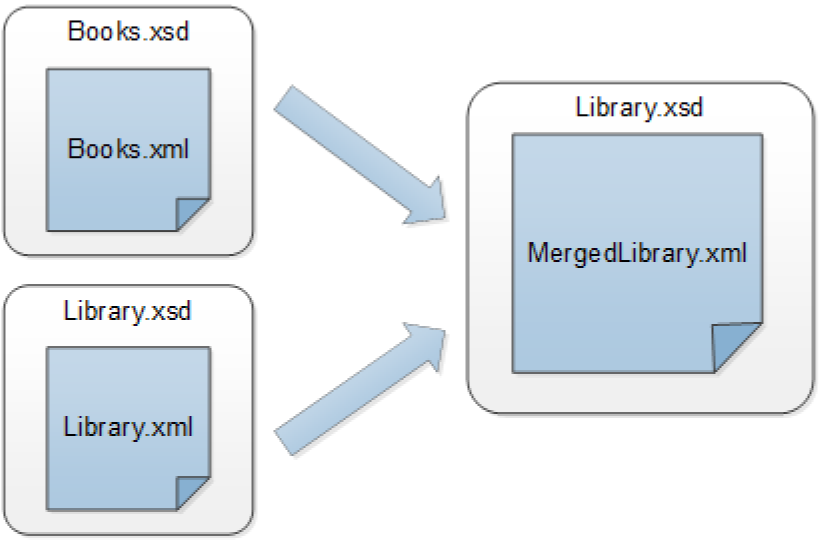
To configure MapForce to write the output directly to final files instead of temporary ones, go to **Tools | Options | General** and select the check box **Write directly to final output files**. Note that enabling this option is not recommended while you follow this tutorial, because you may unintentionally 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 at the bottom of the **Mapping** pane. To generate the XSLT2 code and save it 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 the code generation has been completed, the destination folder will include the following two files:

1. An XSLT transformation file, named after the target schema. This transformation file has the following format: **`MappingMapTo<TargetFileName>.xslt`**.
2. A **`DoTransform.bat`** file, which enables you to run the XSLT transformation with [Altova RaptorXML Server]() from the command line. In order to run the command, you will need to install RaptorXML.

2.2 Multiple Sources to One Target

In this tutorial, you will learn to merge the data from a new file called **`Library.xml`** with the data from **`Books.xml`**. The result will be a target file called **`MergedLibrary.xml`**, which will contain the data from both source files. The target file will be based on the **`Library.xsd`** schema. Note that the source files have different schemas. If the source files had the same schema, you could also merge their data using a different approach, described in Multiple Sources to Multiple Targets 56. The image below represents an abstract model of the data transformation described in this tutorial.



The code listing below shows an extract from **`Books.xml`**, the file that will be used as the first data source.

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

The code listing below shows an extract from **`Library.xml`**, the file that will be used as the second data source:

```
<library>
   <publication>
      <id>5</id>
      <author>Alexandre Dumas</author>
      <title>The Three Musketeers</title>
      <genre>Fiction</genre>
      <publish_year>1844</publish_year>
   </publication>
</library>
```

This is how we want our merged data to look in the target file called **`MergedLibrary.xml`**:

```
<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>
   <publication>
      <id>5</id>
      <author>Alexandre Dumas</author>
      <title>The Three Musketeers</title>
      <genre>Fiction</genre>
      <publish_year>1844</publish_year>
   </publication>
</library>
```

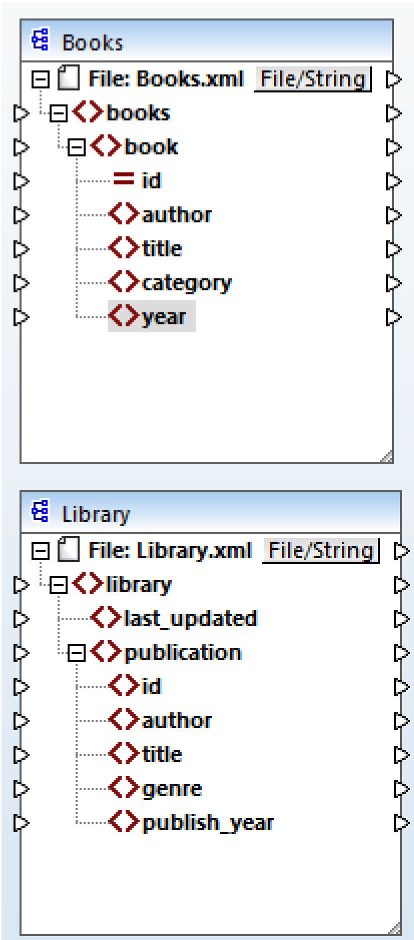
To carry out the transformation, take the steps described in the subsections below.

2.2.1 Prepare Source Files

The starting point of this tutorial is two source files, both of which have a schema (**`Books.xsd`** and **`Library.xsd`**) and data (**`Books.xml`** and **`Library.xml`**). To prepare the source files for the mapping, take the following steps:

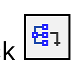
1. Open **`Books.xsd`**.
2. When MapForce suggests adding a sample XML file, click **Browse** and open **`Books.xml`**.
3. Open **`Library.xsd`**.
4. When MapForce suggests adding a sample XML file, click **Browse** and open **`Library.xml`**.
5. For convenience, place the source files one on top of the other (*see screenshot below*).

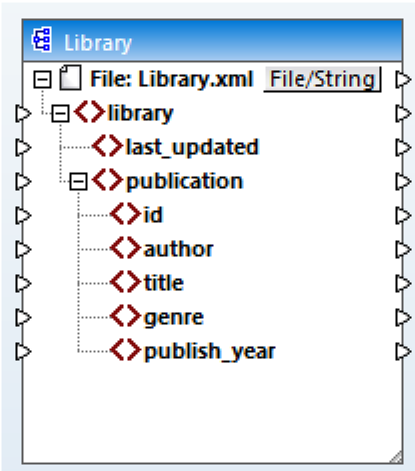
At this stage, our mapping design looks as follows:



As the new mapping references several files from the same folder, make sure to save this new mapping in the **`BasicTutorials`** folder.

2.2.2 Add Target Component

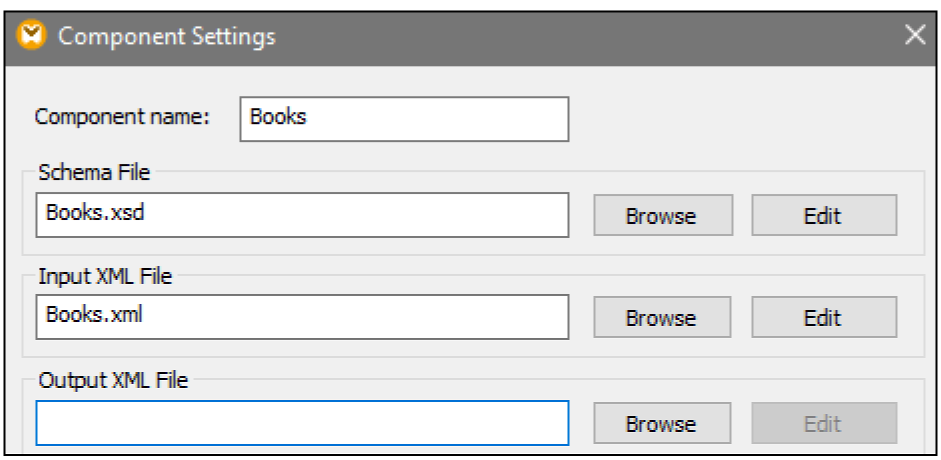
The next step is to add a target file called **`Library.xsd`**. To add the target file, click  (**Insert XML Schema/File**). Click **Skip** when MapForce suggests supplying an instance file. Then click the header of the new component and drag it to the right of the two source components. You can always move mapping components in any direction. Nevertheless, placing a source component to the left of a target component will make your mapping easier to read and understand. This is also the convention for all the mappings illustrated in this documentation and the sample mapping files accompanying your MapForce installation.

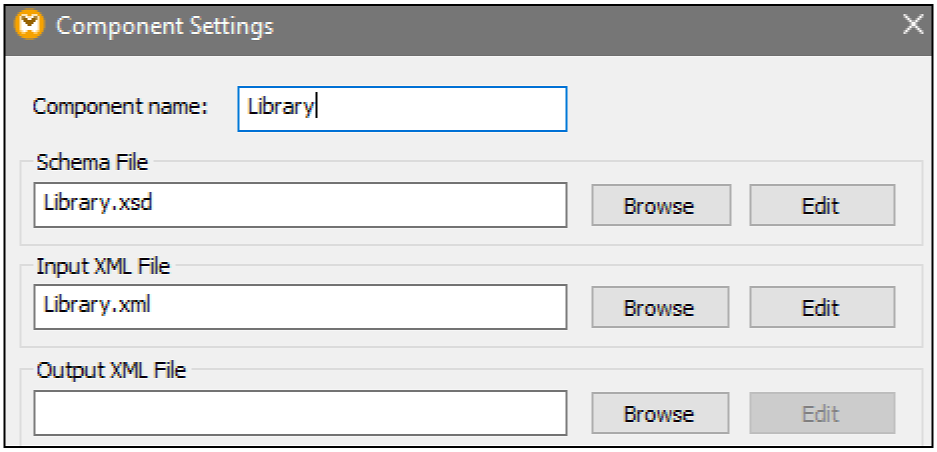


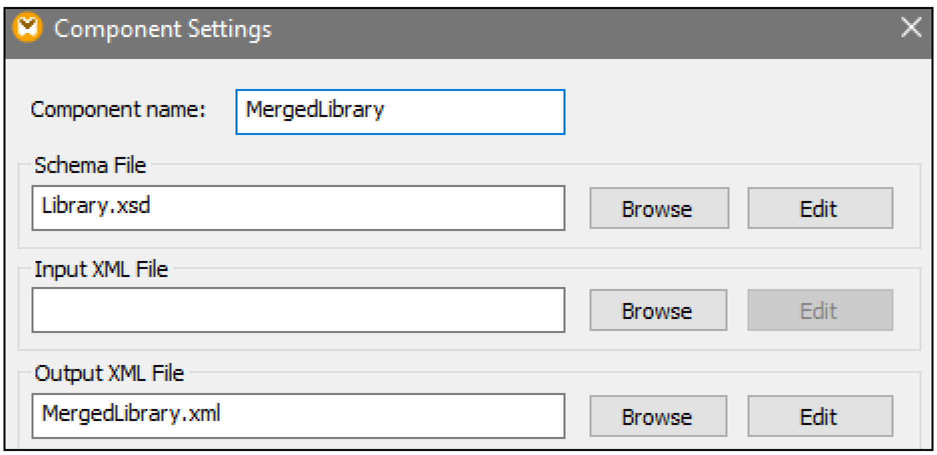
At this stage, the target file only has structure but no data. The data will later be merged from the two source files and mapped to the target file.

2.2.3 Verify and Set Input/Output Files

At his stage, the mapping has two source components (`Books` and `Library`) and one target component (`Library`). Now we have two components with the same name - `Library`. To avoid confusion, we need to change the settings in the Component Settings dialog box 107. Double-click the header of each component. Then verify and change the name and the input/output files of each component as shown below.







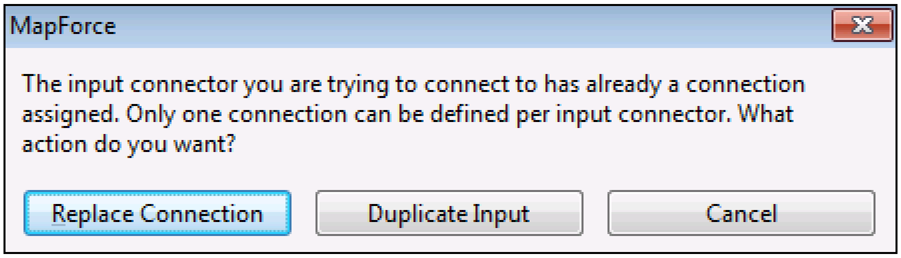
As shown above, the first source component reads data from **`Books.xml`**. The second source component reads data from **`Library.xml`**. Finally, the target component outputs data to a file called **`MergedLibrary.xml`**.

2.2.4 Connect Sources and Target

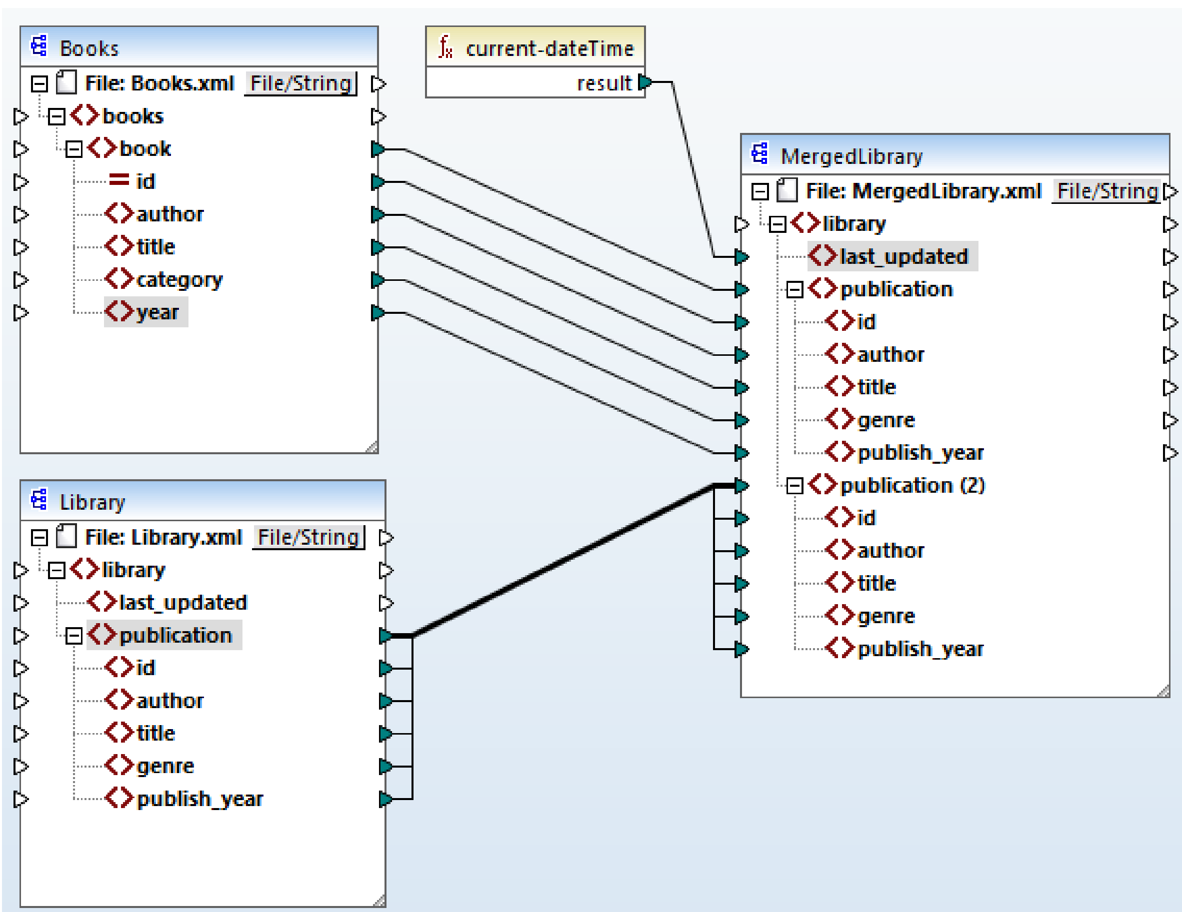
Now we want to map the data from two source files (**`Books.xml`** and **`Library.xml`**) to the target file (**`MergedLibrary.xml`**). To achieve the goal, take the following steps:

- Connect the `<book>` element in the first source component with the `<publication>` element in **`MergedLibrary.xml`**. When you do this, MapForce may automatically connect all the child elements of `<book>` in the source file to the elements with the same names in the target file. In our example, four connections have been created simultaneously. To find out more about the automatic connection of child elements, see the first tutorial[37].
- When you connect the `<book>` element with the `<publication>` element, you will notice that some of the input connectors in the target component have been highlighted in orange. This indicates that these items are mandatory. To ensure the validity of the target XML file, connect the `<category>` element with the `<genre>` element and the `<year>` element with the `<publish_year>` element.

- To supply a value for the `<last_updated>` element, find the function called **`current-dateTime`** in the XSLT2 library. Drag the function to the mapping area and connect `result` with the `<last_updated>` element in **`MergedLibrary.xml`**.
- To instruct MapForce to write data from the second source to the target, press and hold the output connector of the `<publication>` element in **`Library.xml`** and drag it to the input connector of the `<publication>` element in **`MergedLibrary.xml`**. Since the target input connector already has a connection, the following message appears:



In this tutorial, our goal is to map data from two sources to one target. Therefore, click **Duplicate Input**. By doing so, you configure the target component in such a way that it will accept the data from the new source, too. The mapping now looks as follows:



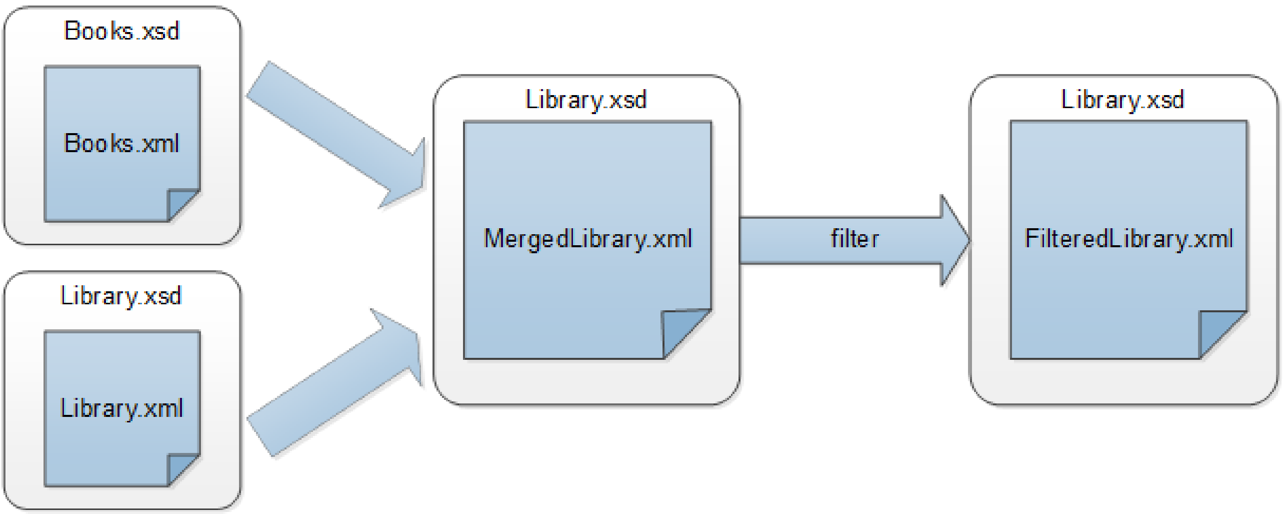
The screenshot above demonstrates that the `publication` item in the target component has been duplicated. The new `publication(2)` node will accept the data from the source component `Library`. Importantly, even though the name of this node appears as `publication(2)` in the mapping, its name in the target XML file will be `publication`, which is our goal in this case.

Click the **Output** button at the bottom of the **Mapping** pane to view the mapping result. You will notice that the data from both **`Books.xml`** and **`Library.xml`** has now been merged into the new **`MergedLibrary.xml`** file. To save the output, open the **Output** pane and select the menu command **Output | Save Output File** or click (**Save generated output**) in the toolbar.

For your convenience, the mapping design in this tutorial is saved as **`Tut2_MultipleToOne.mfd`**. This is useful because this mapping will be used as a starting point in the next tutorial 49.

2.3 Chained Mapping

The goal of this tutorial is to filter the data in **MergedLibrary.xml**, which was created in the previous tutorial [42], in such a way that only a subset of this data is displayed in the output. Specifically, we will only need the books published after 1900. The image below illustrates an abstract model of the data transformation described in this tutorial.

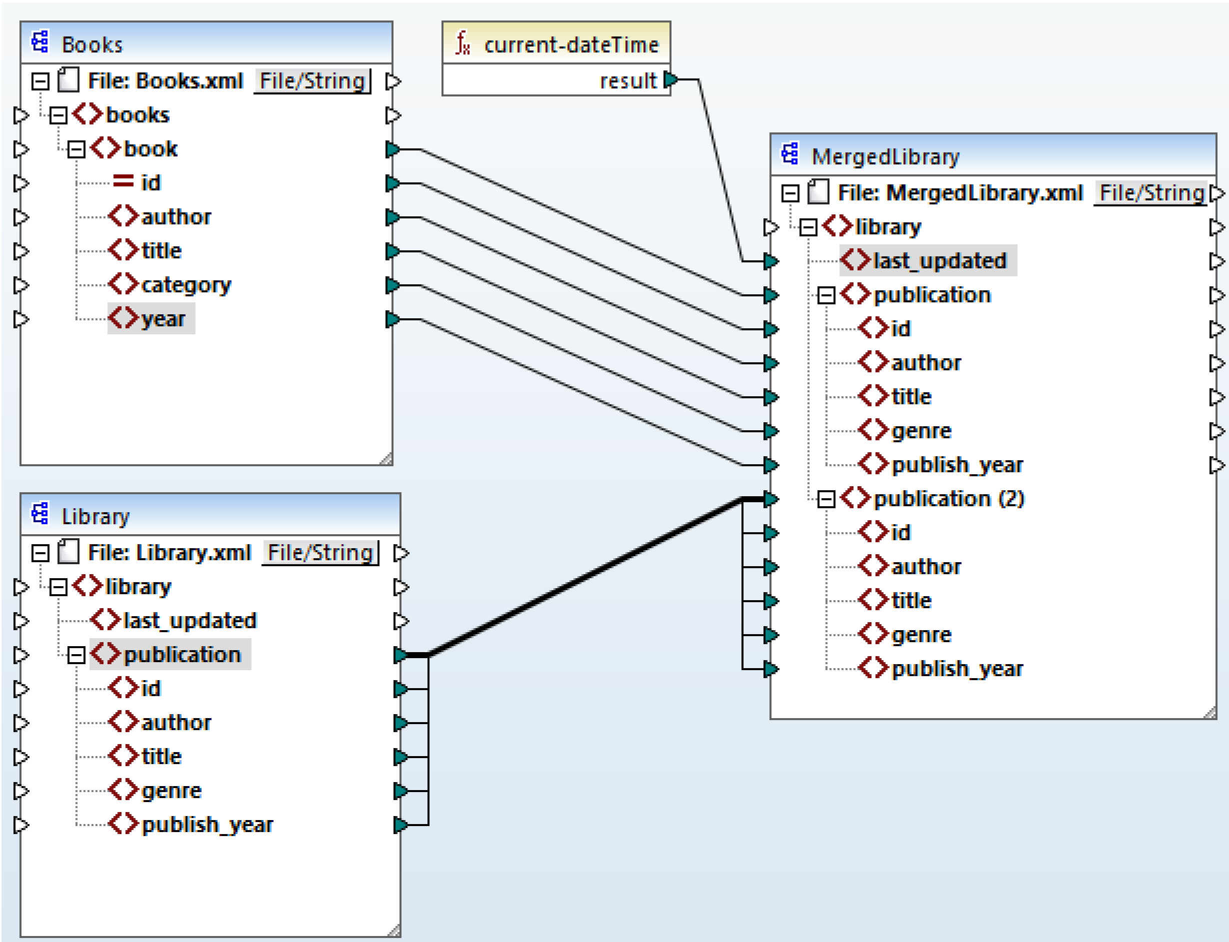


In the diagram above, the data is first merged from two source files (**Books.xml** and **Library.xml**) into a single target file called **MergedLibrary.xml**. Then the data is transformed with a filtering function and passed further to the next component called **FilteredLibrary.xml**. Note that **FilteredLibrary.xml** is based on the **Library.xsd** schema. The intermediate component acts both as a data target and source. In MapForce, this technique is known as *chained mappings* [371].

To carry out the mapping, take the steps described in the subsections below.

2.3.1 Prepare Mapping Design

The starting point of this tutorial is **Tut2_MultipleToOne.mfd** (*see screenshot below*). This mapping was designed in the previous tutorial [42].

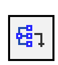


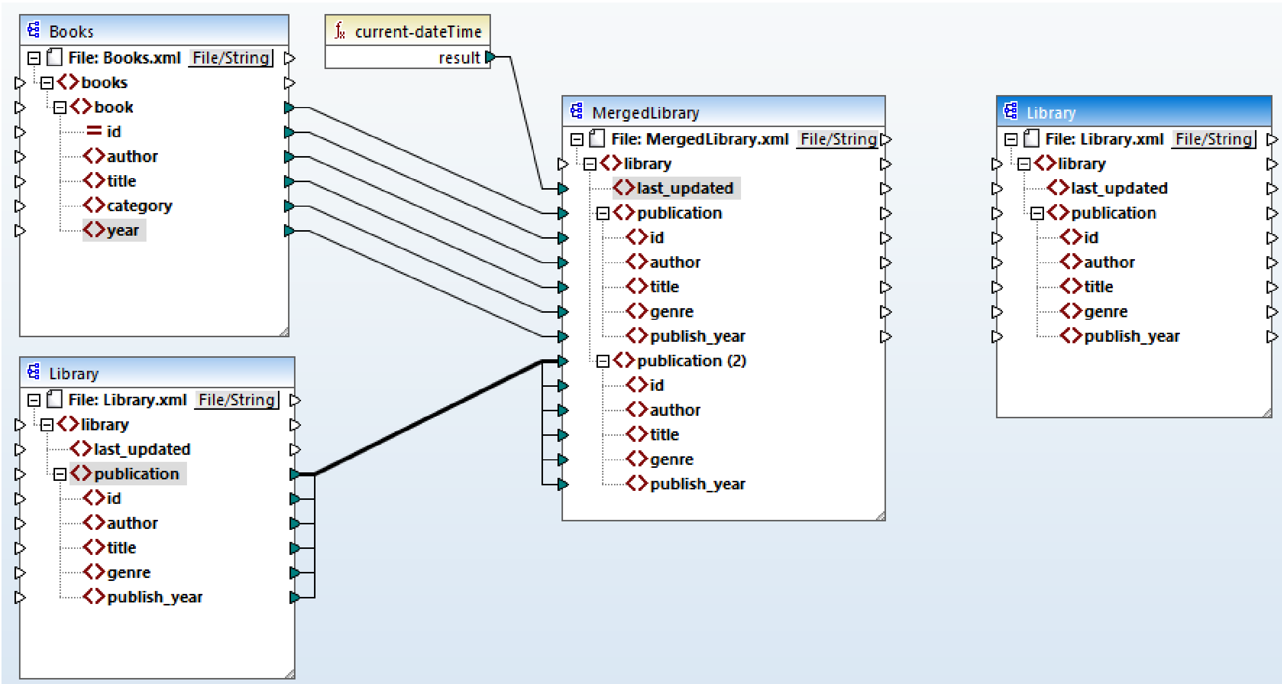
As the new mapping references several files from the same folder, make sure to save this new mapping in the **`BasicTutorials`** folder.

2.3.2 Configure Second Target

Now we need to add and configure the second target file (**`FilteredLibrary.xml`**), which will contain only a subset of <book> elements from **`MergedLibrary.xml`**.

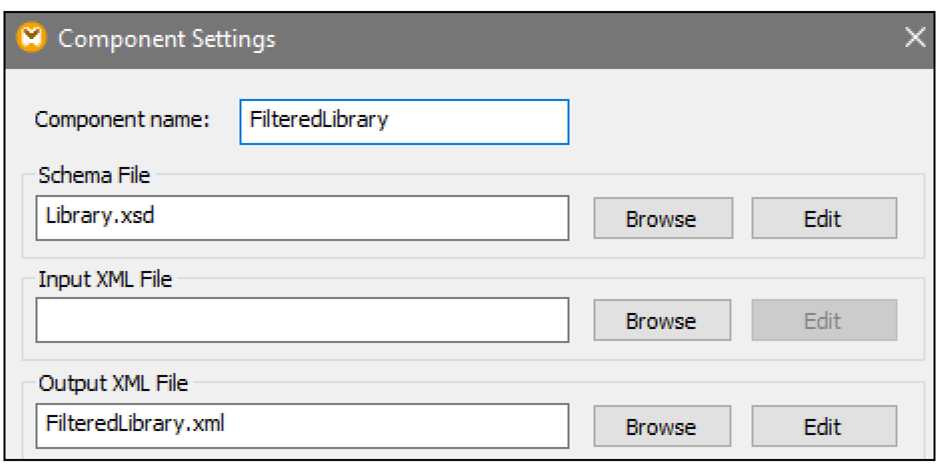
Add the second target component

To add the second target component, click the toolbar button  (**Insert XML Schema/File**) and open **`Library.xsd`**. Click **Skip** when prompted to supply a sample instance file. The second target component has only structure but no content. At a later stage, we will map the filtered data to this target file. The mapping design now looks as follows:



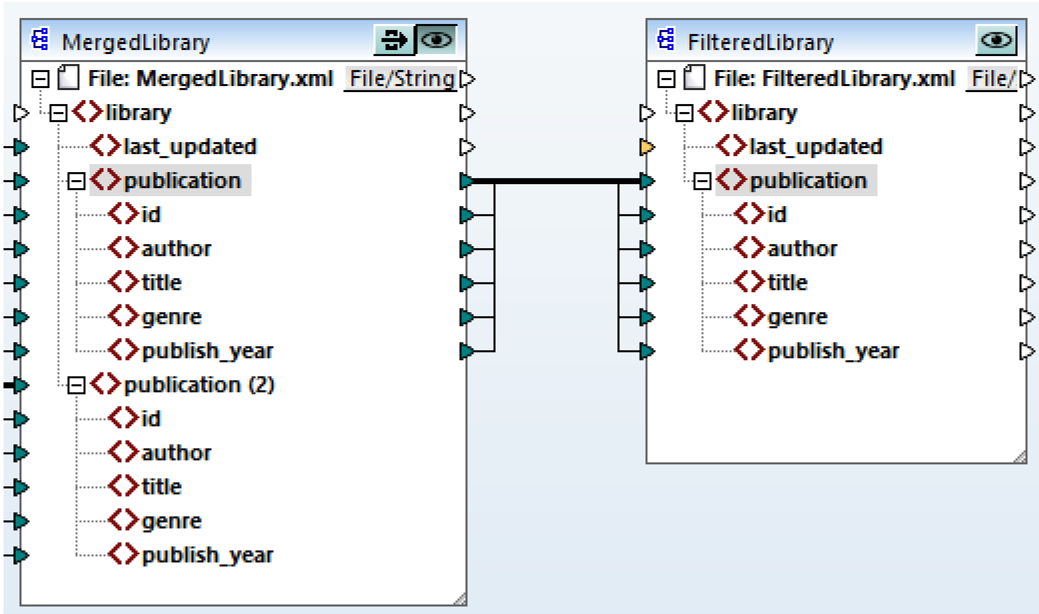
Configure the second target component

As shown above, the mapping now has two source components (`Books` and `Library`) and two target components (`MergedLibrary` and `Library`). To avoid confusion, we will rename the newly added component `FilteredLibrary`. To do this, double-click the header of the right-most component and edit the component settings [107] as follows:



2.3.3 Connect Targets

The next step is to map the `<publication>` element in `MergedLibrary` to the `<publication>` element in `FilteredLibrary`. When you connect the output connector 66 of `MergedLibrary` with the input connector of `FilteredLibrary`, the following notification message is displayed: "You have created multiple target components in the mapping project. To preview the output of a specific target component, click Preview button in the title bar of that component, then click the XSLT, or Output tab, to preview the result." 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 next steps.

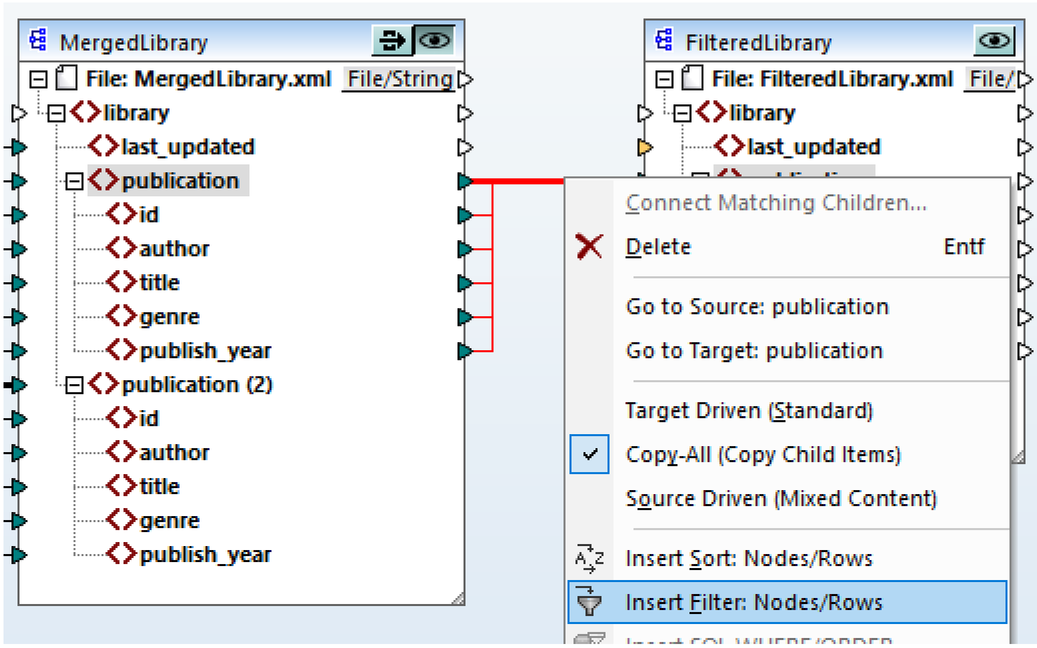


2.3.4 Filter Data

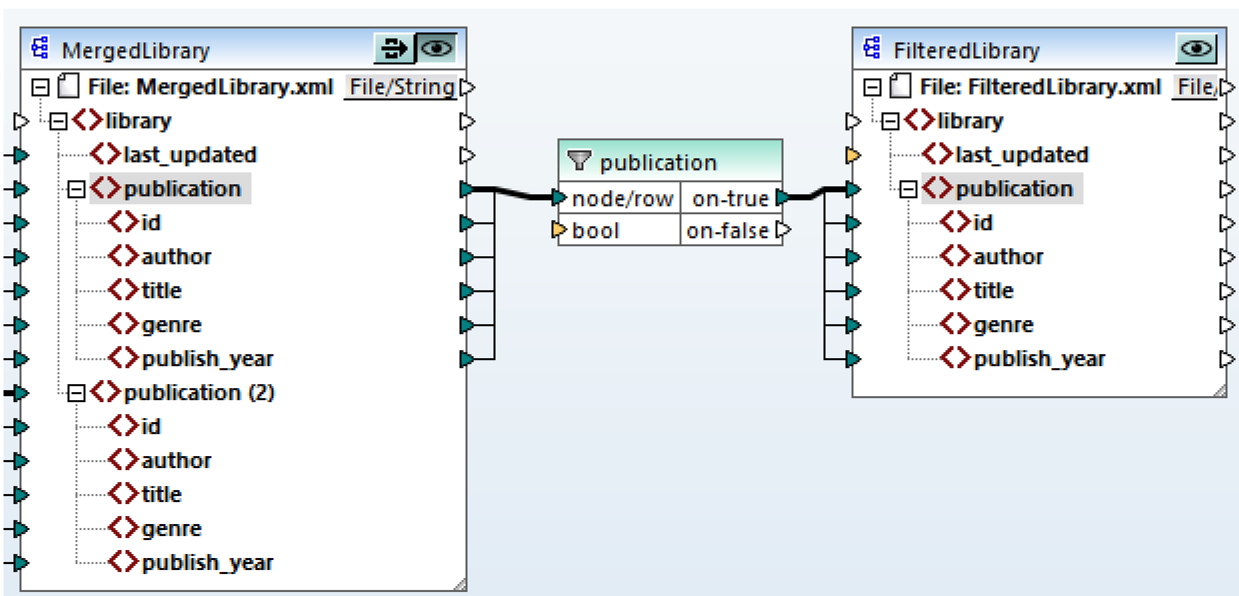
In this step, we will filter the data from `MergedLibrary` in such a way that only the books published after 1900 will be passed to the `FilteredLibrary` component. We will use a **Filter** component for this purpose.

Add a filter

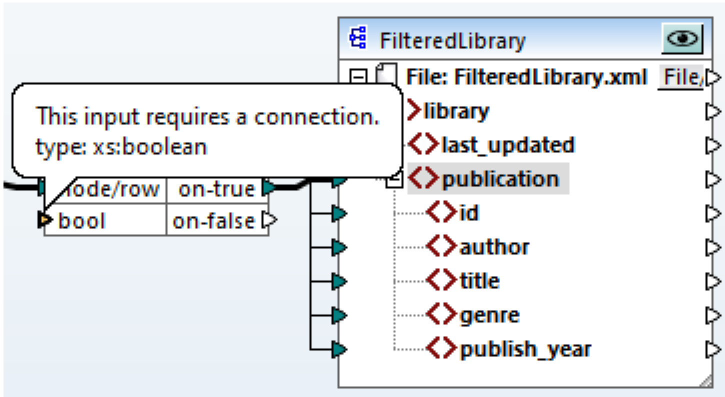
To add a filter, right-click the connection between `MergedLibrary` and `FilteredLibrary` and select **Insert Filter: Nodes/Rows** from the context menu.



The filter component has now been added to the mapping (*see screenshot below*).



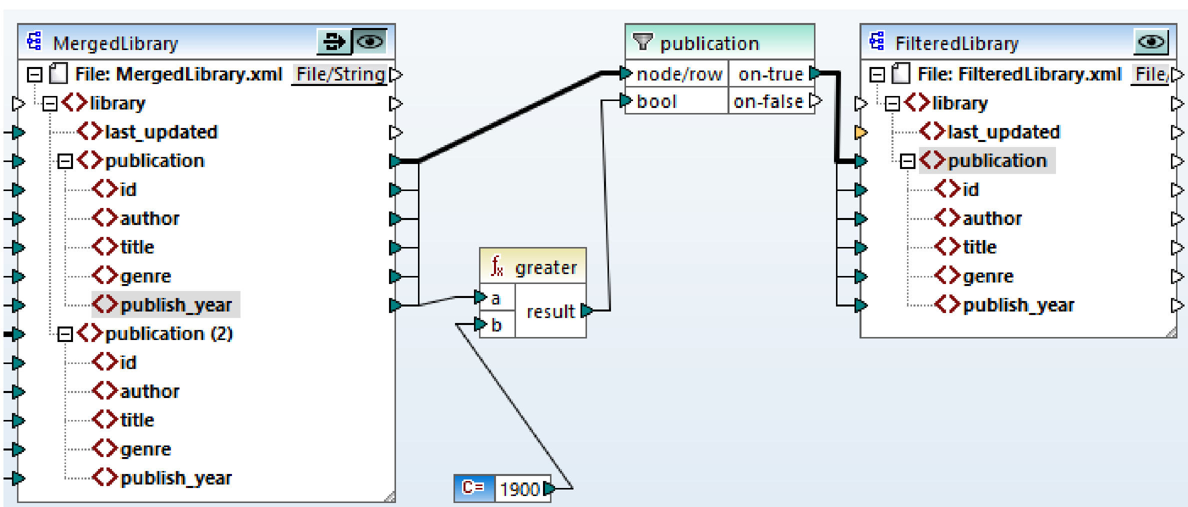
As shown above, the `bool` input connector is highlighted in orange, which means that an input is mandatory. If you hover over the connector, you will see that an input of type `xs:boolean` is required (*see screenshot below*). To see tips, click  (**Show tips**) in the toolbar.



Only books after 1900

The filter component requires a condition that returns either `true` or `false`. When the Boolean condition returns `true`, the data of the current `publication` sequence will be copied over to the target. When the condition returns `false`, the data will not be copied. In this tutorial, the required condition is to filter all the books that were published after 1900. To create the condition, do the following:

1. Click **Constant** in the **Insert** menu. Add a constant with the value *1900*. Choose **Number** as a type.
2. In the **Libraries** window, locate the `greater` function and drag it to the mapping pane.
3. Make the mapping connections to and from the `greater` function, as shown below. By doing this, you are instructing MapForce to copy the current source `<publication>` element to the `<publication>` element in the target component when `publish_year` is greater than 1900.



2.3.5 Preview and Save Output

We are now ready to preview and save the output of each target component. When multiple target components exist in the same mapping, you can choose which one to preview by clicking the button. When the

Preview button is in a pressed state, it indicates that that specific component is currently enabled for preview, and it will generate the output in the **Output** pane. Only one component at a time can have the preview enabled.

Therefore, when you want to view and save the output of the intermediate component `MergedLibrary`, do the following:

1. Click  in the `MergedLibrary` component.
2. Click the **Output** button at the bottom of the **Mapping** pane.
3. In 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 `FilteredLibrary` component:

1. Click  in the `MergedLibrary` component.
2. Click  in the `FilteredLibrary` component.
3. Click the **Output** button at the bottom of the **Mapping** pane.
4. In the **Output** menu, click **Save Output File** if you want to save the output to a file.

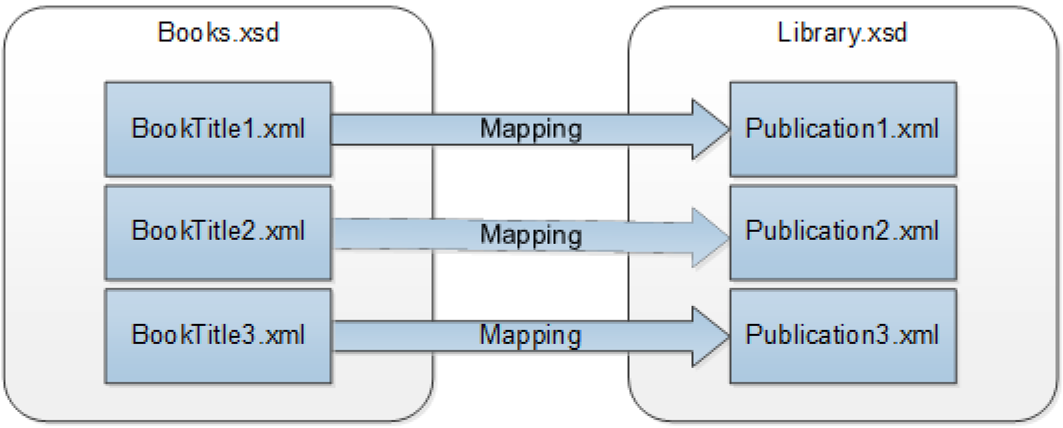
You have now finished designing the mapping which has two target components. For further information about working with pass-through components, see Chained Mappings 371. For your convenience, the mapping design in this tutorial is saved as **`Tut3_ChainedMapping.mfd`**.

2.4 Multiple Sources to Multiple Targets

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

1. To read data from multiple XML files in the same directory.
2. To map the schema of each file to a new schema.
3. For each source XML file, to generate a new XML target file with the new schema.
4. To strip the XML and namespace declaration from the generated files.

The image below illustrates an abstract model of the data transformation used in this tutorial:



Starting point

We will use three source XML files as examples. They are named **`BookTitle1.xml`**, **`BookTitle2.xml`**, and **`BookTitle3.xml`**. Each of the three files is based on **`Books.xsd`** and stores a single book (*see below*).

`BookTitle1.xml`

```
<?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>
```

`BookTitle2.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>
```

BookTitle3.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>
```

Further steps

This is how are going to proceed:

1. First of all, we need to map the source schema (**Books.xsd**) to the target schema (**Library.xsd**).
2. After the transformation, the mapping will generate three files according to this new schema (*see code listings below*).
3. We will also configure the mapping so that the names 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.

   Publication1.xml

   ```
   <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>
   ```

   Publication2.xml

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

   Publication3.xml

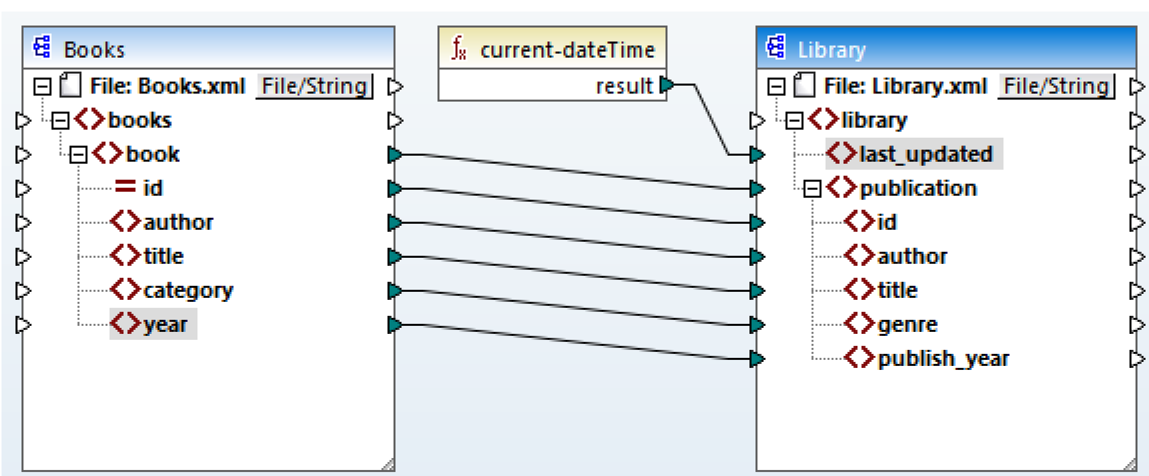
```
<library>
   <publication>
      <id>3</id>
      <author>Herman Melville</author>
      <title>Moby Dick</title>
      <genre>Fiction</genre>
      <publish_year>1851</publish_year>
   </publication>
</library>
```

To carry out the required data transformation, take the steps described in the subsections below.

2.4.1 Prepare Mapping Design

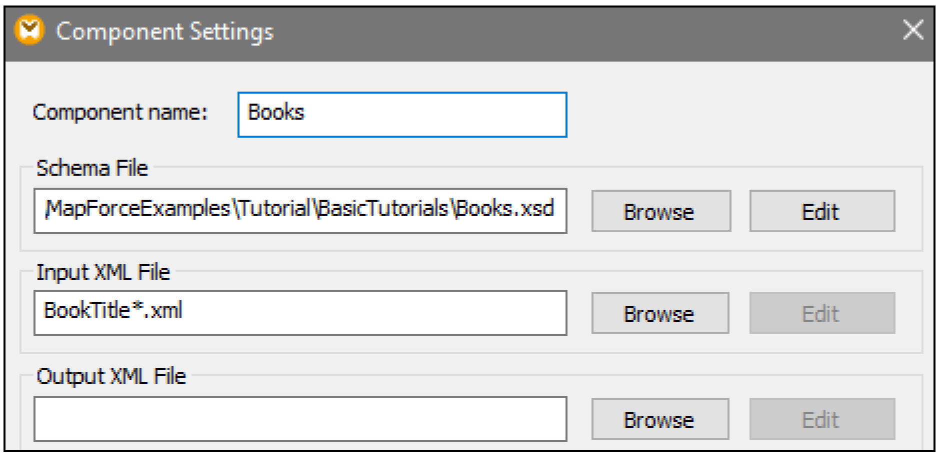
The starting point of this tutorial is similar to the **BooksToLibrary.mfd** mapping from the first tutorial 32, the difference being that no XML files are supplied at this stage. Our goal in this step is to map one abstract structure (**Books.xsd**) to another abstract structure (**Library.xsd**). To reproduce the scenario illustrated in the screenshot below, take the following steps:

1. Open **Books.xsd**.
2. When MapForce suggests adding a sample XML file, click **Skip**.
3. Open **Library.xsd**.
4. When MapForce suggests adding a sample XML file, click **Skip**.
5. Connect the respective nodes, as shown in the screenshot below.
6. Locate the **current-dateTime** function in the **Libraries** window.
7. Drag the function to the mapping area and connect `result` with the `last_updated` element.



2.4.2 Configure Input

Now we want three XML files, each containing one book, to be based on the same schema called **`Books.xsd`**. These XML files, called **`BookTitle1.xml`**, **`BookTitle2.xml`**, and **`BookTitle3.xml`**, will be used as input files. Before changing the component settings, save your mapping as an **`.mfd`** file in the **`BasicTutorials`** folder. To instruct MapForce to process multiple XML instance files, double-click the header of the source component. Enter **`BookTitle*.xml`** as an input file in the Component Settings dialog box [107]. The asterisk (`*`) in the file name instructs MapForce to use all the files with the `BookTitle` prefix as mapping inputs. Because the path is relative, MapForce will look for all `BookTitle` files in the same directory as the mapping file. You can also enter an absolute path if necessary.

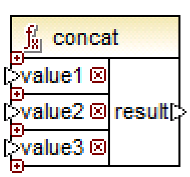


2.4.3 Configure Output Part 1

The next step is to create the file name of each output file. For this purpose, we will use the **`concat`** function, which concatenates (joins) all the values supplied to it as arguments. When these values are joined together, they will create an output file name (e.g., **`Publication1.xml`**). To generate the file names using the **`concat`** function, take the steps described below.

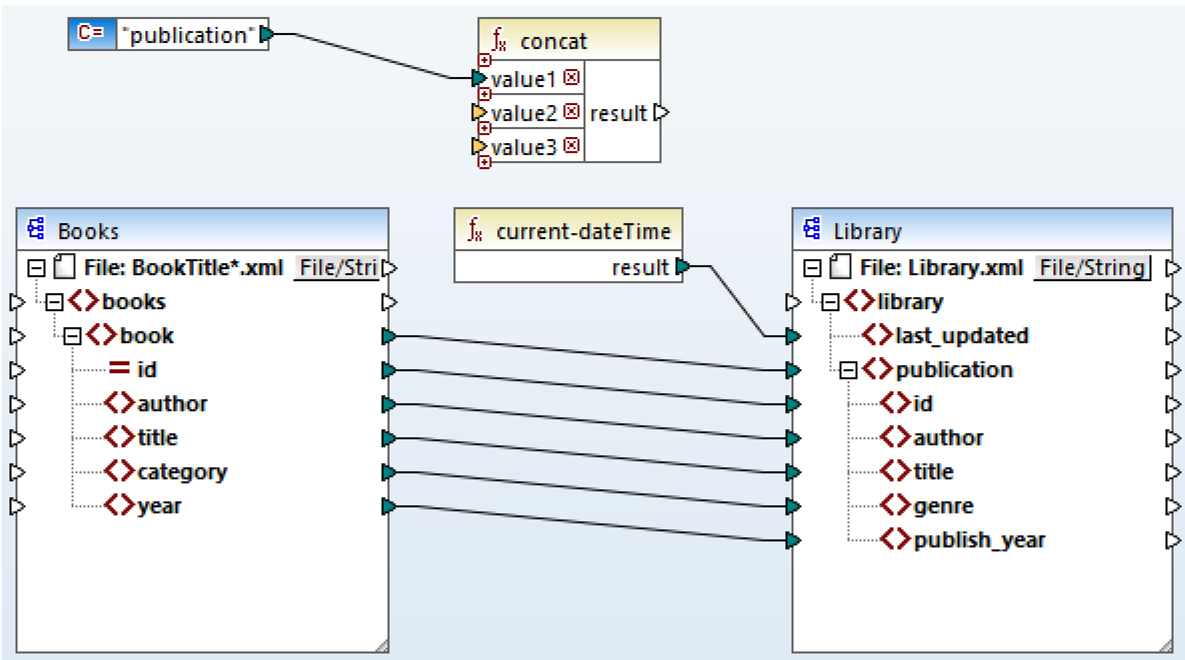
Add the concat function

Search for the **`concat`** function in the **Libraries** window and drag it to the mapping area (*see screenshot below*). By default, this function has two parameters when it is added to the mapping. In our example, we need three parameters. Click (**Add parameter**) inside the function component and add a third parameter to it. Note that clicking (**Delete parameter**) deletes a parameter.



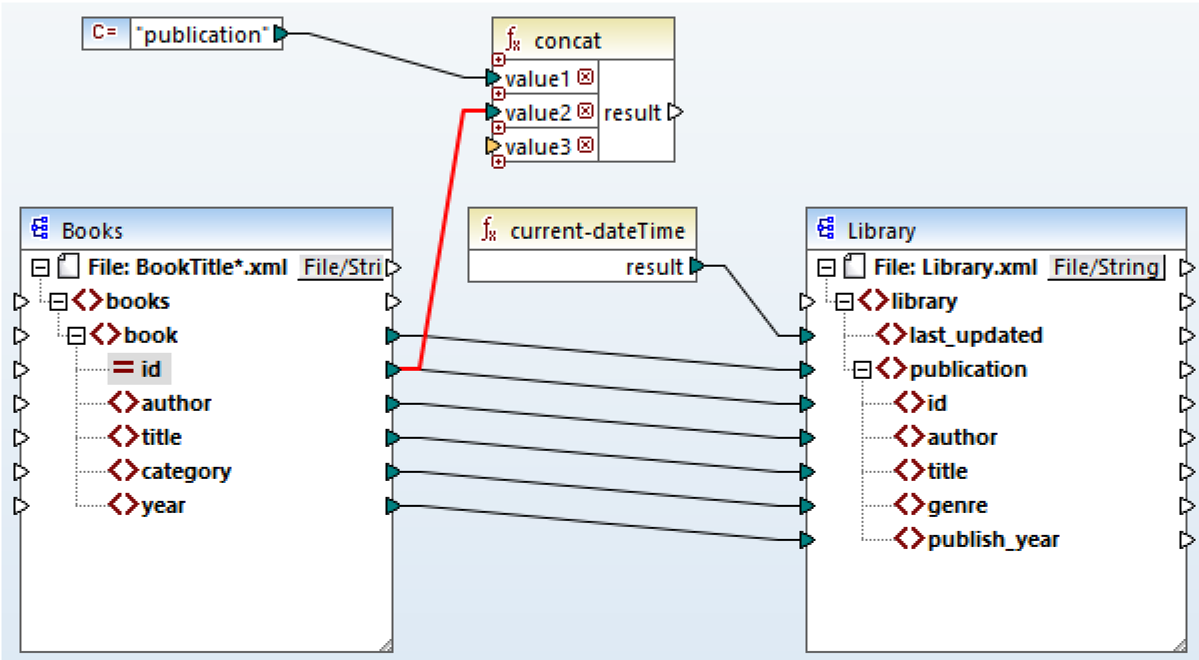
Insert a constant

To add a constant, click **Constant** in the **Insert** menu. When you are prompted to supply a value, enter `publication` and leave the **String** option unchanged. The constant `publication` supplies the constant string value `publication`. Connect the constant with `value1` of the `concat` function, as shown in the screenshot below:



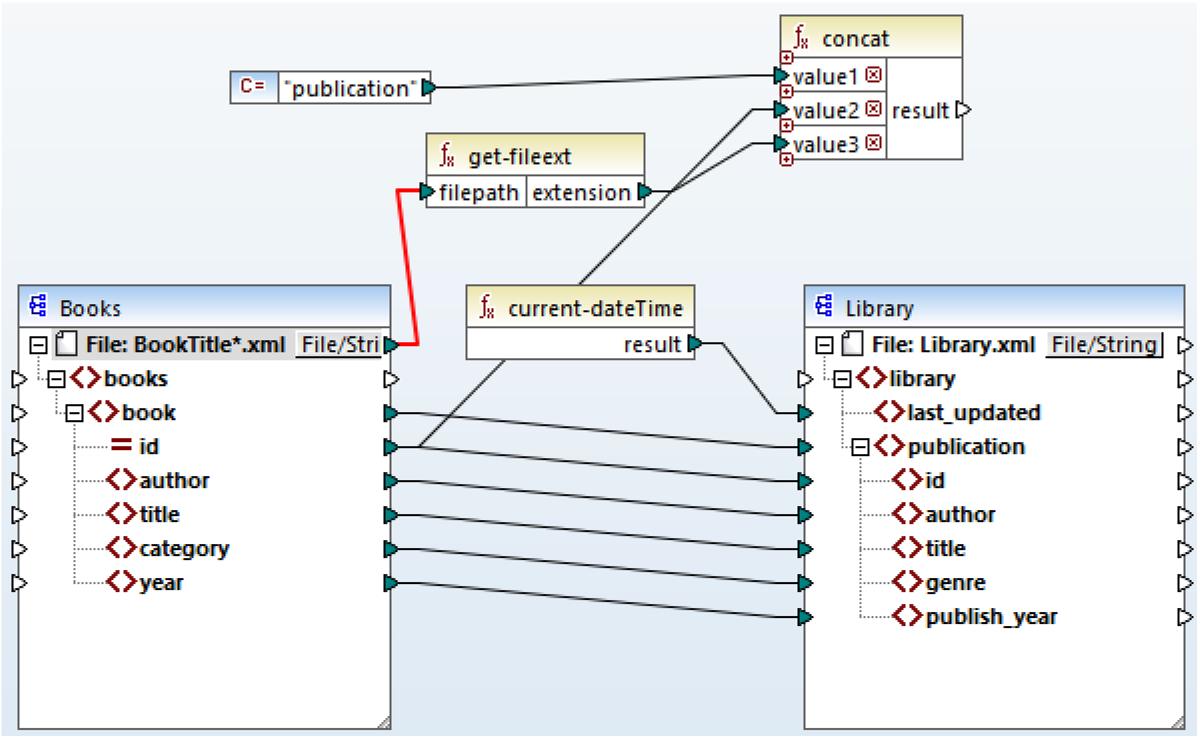
Supply the id

Connect the `id` attribute of the source component with `value2` of the `concat` function. The attribute `id` 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. The connection becomes red when you click on it.



Extract the file extension

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: BookTitle*.xml`**) to the `filepath` parameter of this function. Then connect the `extension` parameter 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 and pass it to the output file name.

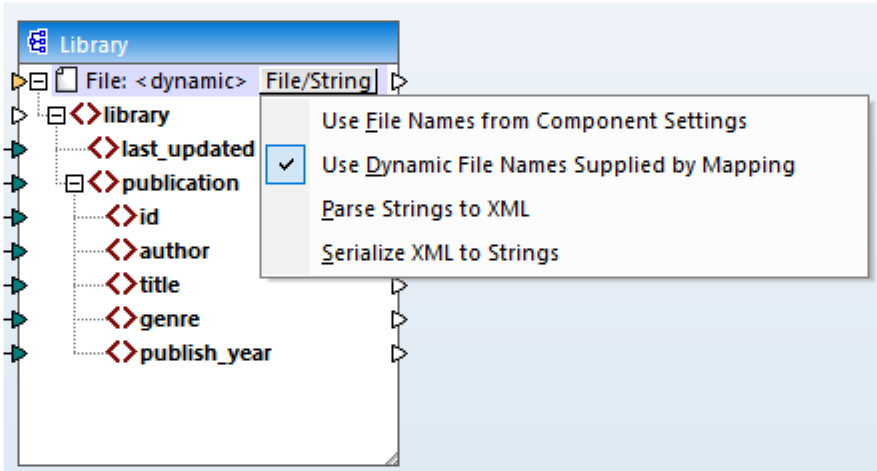


2.4.4 Configure Output Part 2

We can now instruct MapForce to create the file names when the mapping runs. In order to do this, we will use dynamic file names (*see subsections below*).

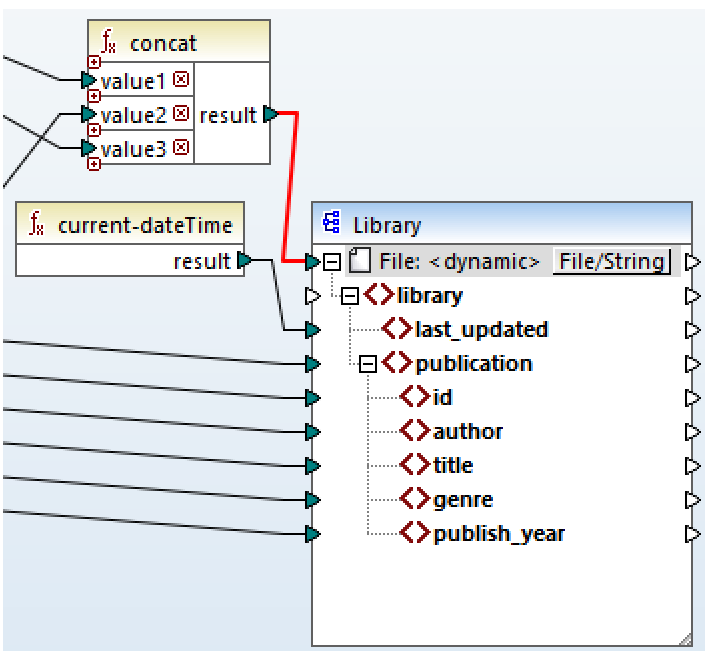
Dynamic file names

At this stage, we need to instruct MapForce to generate the instance files dynamically, which means that the every output file will receive its name based on the arguments supplied to the `concat` function. To do this, click File or File/String of the target component and select **Use Dynamic File Names Supplied by Mapping**.



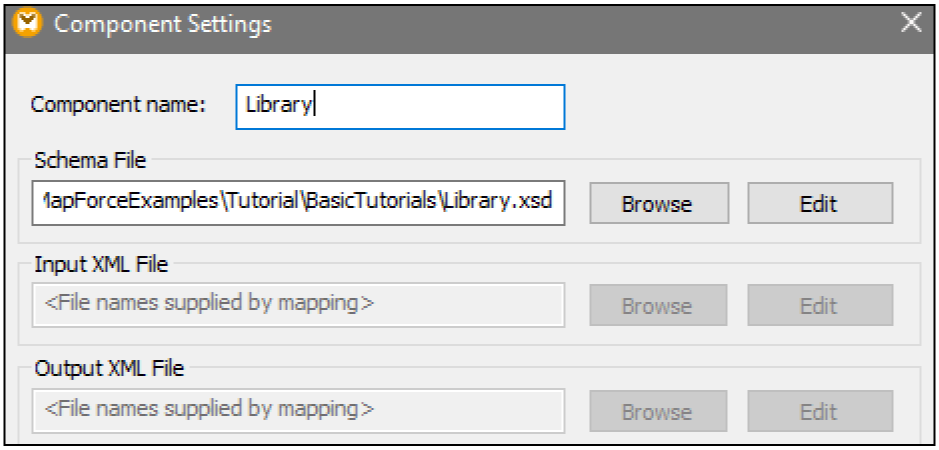
Connect the function with the dynamic node

The next step is to connect the result of the **concat** function with the `File: <dynamic>` node of the target component.

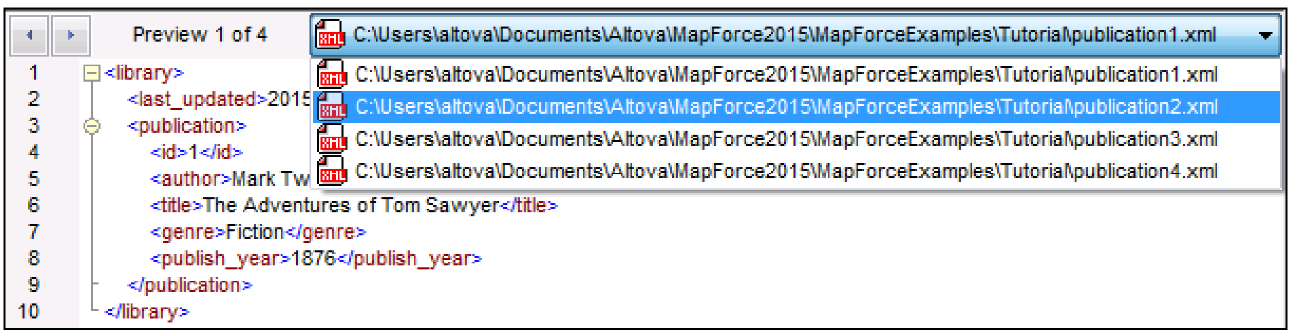


Check the component settings

If you double-click the header of the target component 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>* (*see screenshot below*). This indicates that you have supplied the instance file names dynamically from the mapping. Therefore, it is no longer relevant to define them in the component settings.



You can now run the mapping and see the result as well as the names of the 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 (*see screenshot below*).

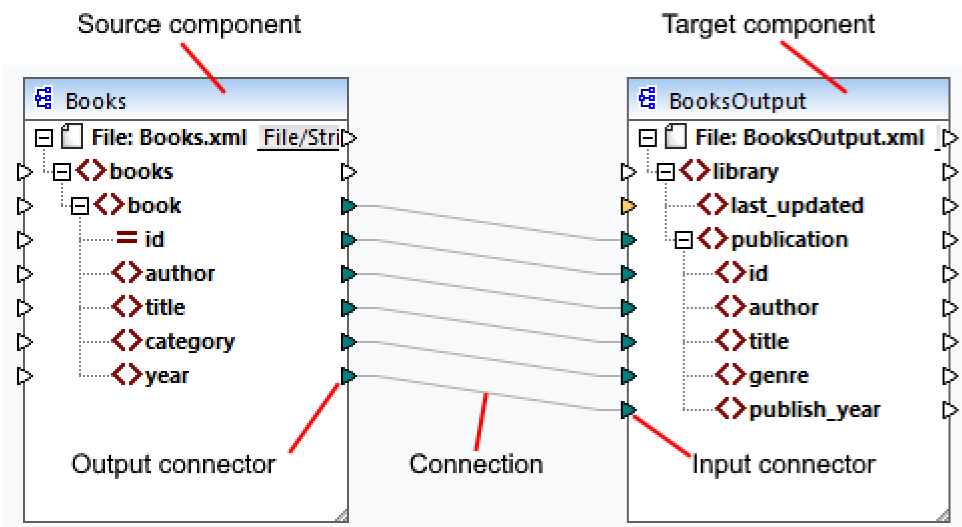


For your convenience, the mapping design in this tutorial is saved as `Tut4_MultipleToMultiple.mfd`.

3 Mapping Fundamentals

A MapForce mapping design (or simply mapping) is the visual representation of how data is to be transformed from one format into another. A mapping consists of *components* that you add to the mapping area to create your data transformations. 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 this mapping execution using MapForce Server or FlowForce Server, for example. MapForce saves mappings as `.mfd` files.

The screenshot below illustrates the basic structure of a mapping:



New mapping

To create a new mapping, click (**New**) in the toolbar. Alternatively, click **New** in the **File** menu. The next step is to add components [69] to the mapping and create connections [78].

Main parts of a mapping

The subsections below describe the main parts of a mapping design.
Component
In MapForce, the term *component* is what represents visually the structure of your data, or how data is to be transformed. Components are the central building pieces of any mapping and are represented as rectangular boxes. Components can be divided into two large groups:

- Source and target components;
- Structural [105] and transformation [138] components.

Note that these two groups are not mutually exclusive. The first group reflects the relations between components: e.g., one component can be a source for one component and a target for another component. MapForce reads data from a source component and writes this data to a target component. When you run a mapping, the target component instructs MapForce to generate a file (or multiple files) or output the result as a string value for further processing in an external program. The types of components from the first group are described below:

- A *source* is located on the left of a target component. MapForce reads data from the source.
- A *target* is located on the right of a source. MapForce writes data to the target component.
- A *pass-through* component is a subtype of source and target components. A pass-through component acts both as a source and target. For more information, see Chained Mappings [371]. Note that only structural components can be pass-through.

The second group shows whether a component has a data structure or is used to transform data mapped from another component.

To find out more about components and component-related actions, see Components [67].

Connector

A connector is a small triangle located on the left or right side of a component. Input connectors are on the left of a component and show data entry points *to that component*. Output connectors are on the right of a component and show data exit points *from that component*.

Connection

A connection is a line that you can draw between two connectors. When you create connections, you instruct MapForce to transform data in a specific way: for example, to read data from an XML document and write it to another XML document.

In this section

This section describes the most common MapForce tasks and concepts. The section is organized into the following subsections:

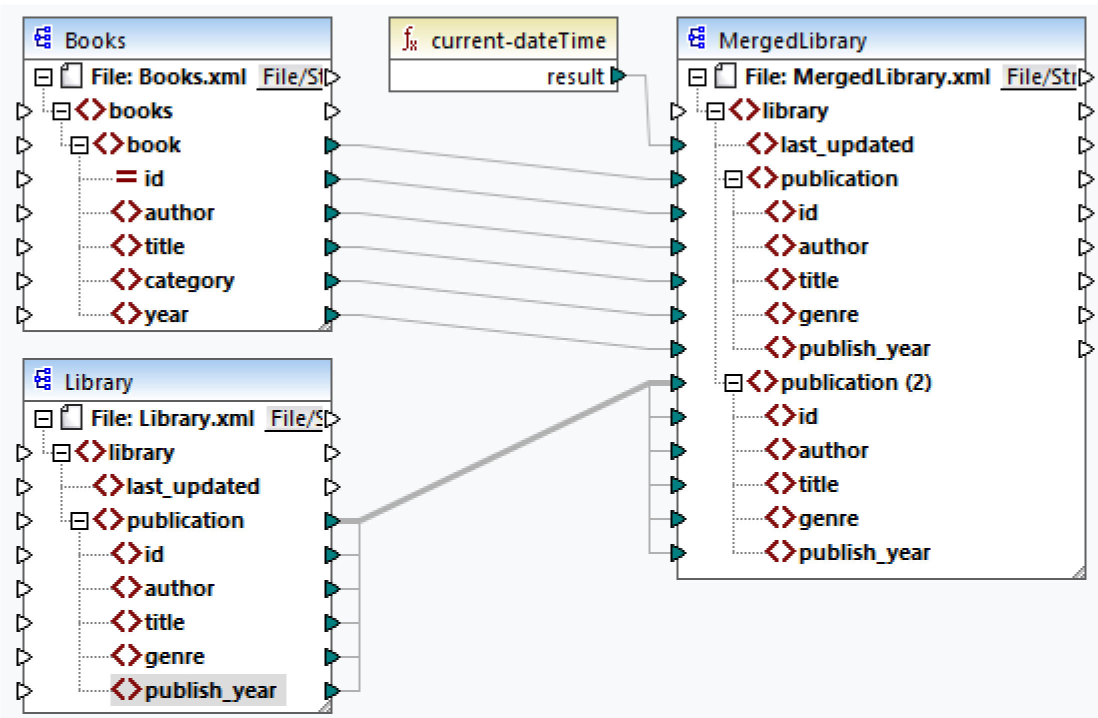
- Components [67]
- Connections [78]
- General Procedures and Features [94]

3.1 Components

Components are the central elements of any mapping design in MapForce. Visually, components are represented as rectangular boxes in the mapping area. This topic gives an overview of structural and transformation components (*see example below*). The distinction is based on whether a component has a data structure or is used to transform data. See the description of these types in the subsections below. See also Mapping Fundamentals[65].

Components example

The sample mapping below illustrates two data source components (`Books` and `Library`), one data target component (`MergedLibrary`), and one transformation component (the **`current-dateTime`** function).



Structural components

Structural components represent an abstract structure of your data (e.g., an XML file). The list of structural components that can be used as data sources and targets is given in Structural Components[105]. Structural components can read data from files or other sources, write data to files or other sources, and store data at some intermediary stage in the mapping process (e.g., to preview the data). The table below gives an overview of structural components and their respective toolbar buttons.

Icon	Description
	XML component
	Text component (*Professional and Enterprise editions*)

Icon	Description
	Database component (*Professional and Enterprise editions* for SQL Databases; *Enterprise Edition* for NoSQL databases)
	JSON component (*Enterprise Edition*)
	Microsoft Excel component (*Enterprise Edition*)
	WSDL component (*Enterprise Edition*)
	EDI component (*Enterprise Edition*)
	XBRL component (*Enterprise Edition*)
	Protocol Buffers (*Enterprise Edition*)

Transformation components

Transformation components help you transform data [190], store an intermediate mapping result [150] for further processing, replace a value by another value [174], sort [162], group [184], and filter [168] your data. The table below gives an overview of transformation components and their respective toolbar buttons.

Icon	Description
	Simple input
	Simple output
	Filter component
	Sort component
	Built-in function
	User-defined function
	SQL/NoSQL-WHERE/ORDER component (*Professional and Enterprise editions*)
	Value-Map component
	Variable
	Web service function (*Enterprise Edition*)
	Exception (*Professional and Enterprise editions*)
	Constant
	If-Else Condition
	Join component (*Professional and Enterprise editions*)

In this section

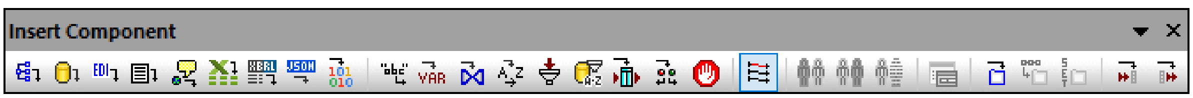
This section gives an overview of components and is organized into the following topics:

- Add Components [69]
- Component Basics [71]
- File Paths [73]

3.1.1 Add Components

This topic explains how to add components to a mapping. To add a component, you need to create a new mapping design [65] first and then do one of the following:

- In the **Insert** menu, choose a component type (e.g., **XML Schema/File**).
- Drag a file from Windows File Explorer into the mapping area. Note that this operation is only possible for compatible file-based components.
- Click the relevant button in the **Insert Component** toolbar (*see screenshot below*).



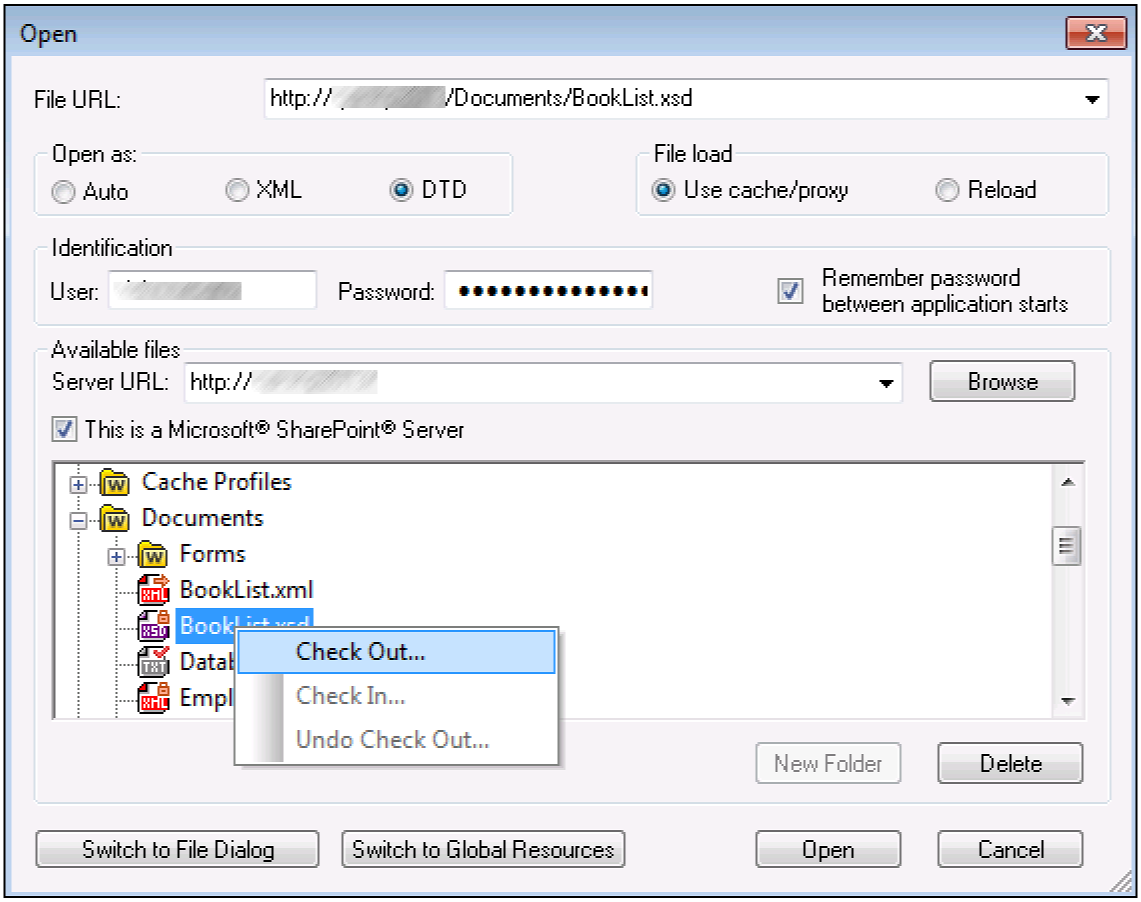
Each component type has a specific purpose and function. To get an overview of components, see Components [67]. To find out more about data structures that can be used as sources and targets, see Structural Components [105]. For information about MapForce built-in components used to store data temporarily or to transform it, see Transformation Components [138].

For a complete list of components, see the component icon reference below.

Add components from URL

Besides adding local files as mapping components, you can also add files from a URL. Note that this operation is supported when you add a file as a source component [65]. The supported protocols are HTTP, HTTPS, and FTP. To add a component from a URL, take the following steps:

1. In the **Insert** menu, select the component type you wish to add (e.g., **XML Schema/File**).
2. In the **Open** dialog box (*screenshot below*), 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.



The list below describes the available options in the **Open** dialog.

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

- *Open as:* Defines the grammar for the parser. The default and recommended option is **Auto**.

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

- *Server URL:* 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 to choose to open the same file type as in Step 1 above. Otherwise, errors will occur.

- *Check in/out:* If you use a Microsoft SharePoint Server, select the check box **This is a Microsoft SharePoint Server**. This 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, right-click the file and select **Check Out** (*see screenshot above*). To check in any file that you previously checked out, right-click the file and select **Check In**.

3.1.2 Component Basics

This topic explains how to set, search and manipulate structural components [67]. For more information, see the subsections below.

Change component settings

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

- Double-click the component header.
- Select the component and click **Properties** in the **Component** menu.
- Right-click the component header and click **Properties**.

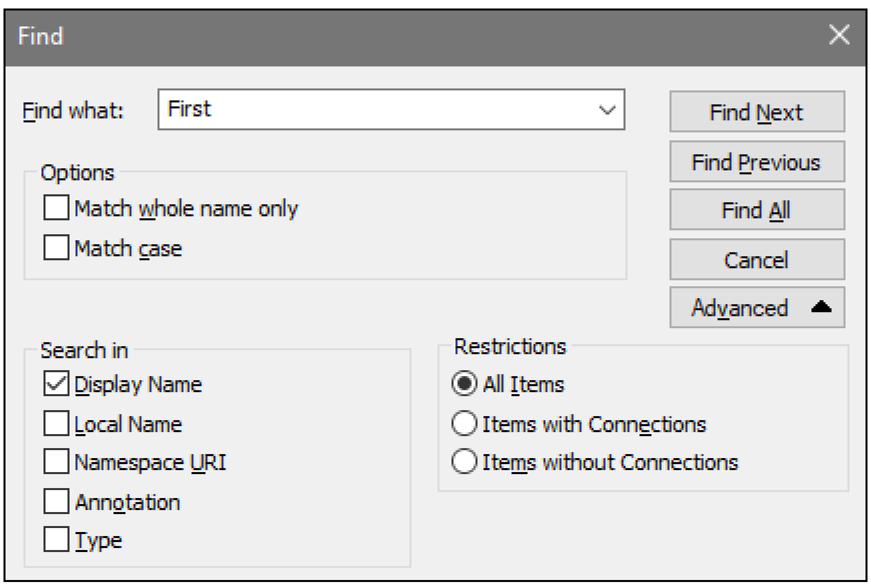
See the list of available settings in XML Component Settings [107].

For any file-based component (e.g., an XML file) the File (*Basic Edition*) or File/String (*Professional and Enterprise editions*) button appears next to the root node. This button specifies advanced options for processing or generating multiple files in a single mapping. For more information, see Processing Multiple Input or Output Files [417].

Search a component

To search for a specific node in a component, take the following steps:

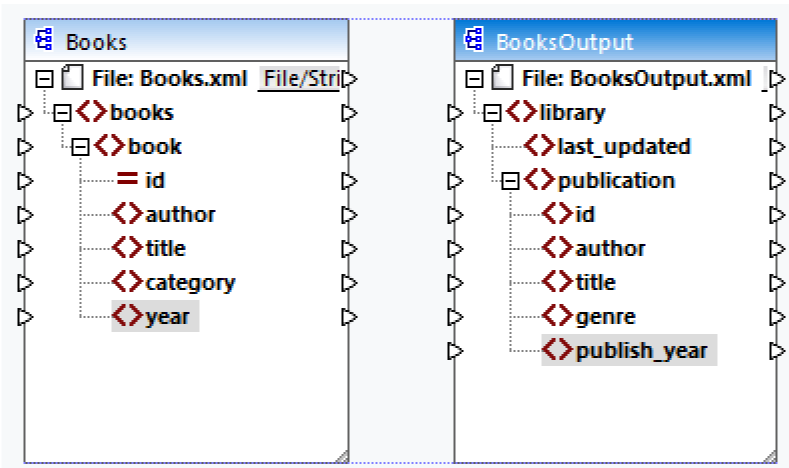
1. Click the component you want to search and press **CTRL+F**.
2. Enter a search term and click **Find Next/Previous/All** (*see screenshot below*).



Use the **Advanced** options to define which items (nodes) are to be searched. You can also restrict the search options based on specific connections.

Align components

When you move components in the mapping area, MapForce shows guide lines (dotted lines) that help you align components (*see screenshot below*).



To enable this option, take the steps below:

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

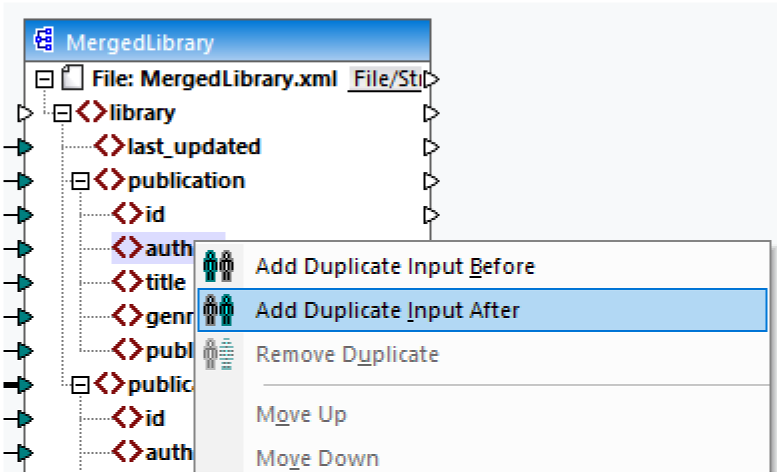
Duplicate input

Sometimes, you may need to configure a component so that it can accept data from more than one source. If you want the target schema to accept data from more than one source schema, you can duplicate any input nodes in your target component. Duplicating input is meaningful only for a target component: Duplicated nodes can only accept data, but it is not possible to map data from duplicated nodes. You can duplicate as many nodes as you need.

There are two ways of duplicating input: (i) selecting **Add Duplicate Input Before/After** from the context menu and (ii) connecting a source node with a target node that is already connected to a different node. The first option is described below. Information about the second option can be found in the second tutorial 42.

Add Duplicate Input Before/After

To duplicate a particular input node, right-click it and select **Add Duplicate Input Before/After** from the context menu (*see screenshot below*). In the image below, the `author` node is about to be duplicated so that data can be mapped to the duplicated node from another source node.



Note: Duplication of XML attributes is not allowed, as it will make the XML instance invalid.

3.1.3 File Paths

A mapping design (**`*.mfd`**) may have references to several schema and instance files. MapForce uses the schema files to determine the structure of the data to be mapped. In MapForce Professional and Enterprise editions, mappings can also include references to StyleVision Power Stylesheet (**`*.sps`**) files, which are used to format data for outputs such as PDF, HTML and Word. Mappings can also have references to file-based databases such as Microsoft Access or SQLite.

References to files 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 on how to set or change paths to different file types referenced by a mapping. The section is organized into the following topics:

- Relative and Absolute Paths [73]
- Paths in Execution Environments [76]

3.1.3.1 Relative and Absolute Paths

This topic explains how to use absolute and relative paths of the files referenced by your component. An absolute path shows the full location of a file, starting with the root directory (*see Example: XML Component below*). A relative path shows the file location which is relative to the current working directory: e.g., **`Books.xml`**.

In the **Component Settings** dialog box (*see example below*), you can specify absolute or relative paths for various files which can be referenced by the component. The list of these files is given below:

- Input files from which MapForce reads data;
- Output files to which MapForce writes data;
- Schema files, which are applicable to components with a schema;

- Structural files, which are used as input or output parameters of user-defined functions and variables;
- StyleVision Power Stylesheet (`*.sps`) files, which are used to format data for outputs such as PDF, HTML and Word;
- Database files in the case of database components (*Professional and Enterprise editions*).

Copy-paste and relative paths
When you copy a component from a mapping and paste it into another mapping, MapForce checks whether the relative paths of schema files can be resolved against the folder of the destination mapping. If the paths cannot be resolved, you will be prompted to make the relative paths absolute.

Broken paths
When you add or change a file reference in a mapping, and the path cannot be resolved, MapForce displays a warning message. Broken path references may happen in the following cases:

- 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.

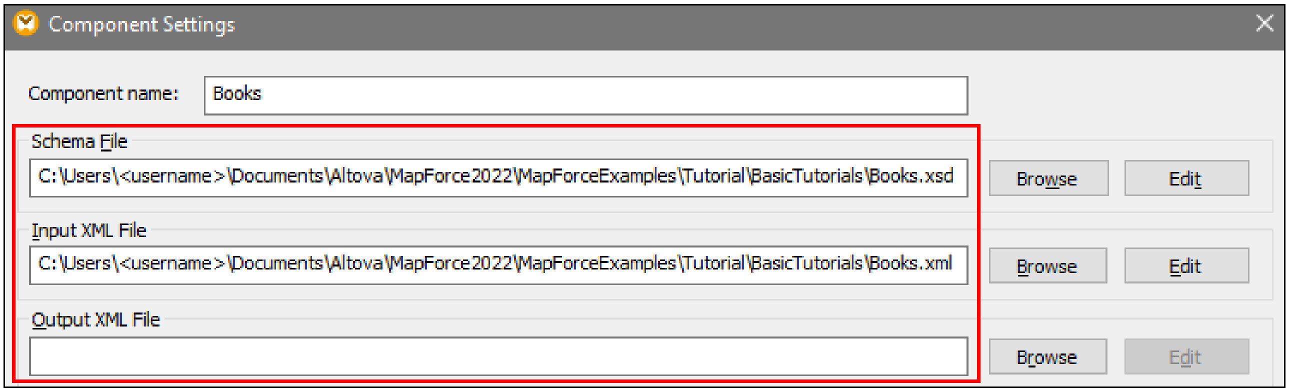
When one of these scenarios happens, MapForce highlights the component in red. The solution is to double-click the red component header and update any broken path references in the **Component Settings** dialog box. See also Change Component Settings 71.

Example: XML component

The example below shows how file paths are used in an XML component. If you want to save all the mapping-related files relative to the mapping file (`.mfd`), check the box *Save all file paths relative to MFD file* at the bottom of the **Component Settings** dialog box. This is the default and recommended option that affects all the files referenced by the component (*shown in the red frame in the image below*). If you have not saved your mapping yet, you will see absolute paths to the schema and/or instance files in the **Component Settings** dialog box. To see relative paths in the **Component Settings** dialog box, take the following steps:

1. Create a new mapping 65 and add a structural component 69: e.g., an XML file with an XML schema assigned to it.
2. Double-click the header of the component to open the **Component Settings** dialog box.
3. Check the box *Save all file paths relative to MFD file* at the bottom of the **Component Settings** dialog box.
4. Save your mapping.
5. You can now open **Component Settings** again to check the relative paths in the relevant text fields.

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



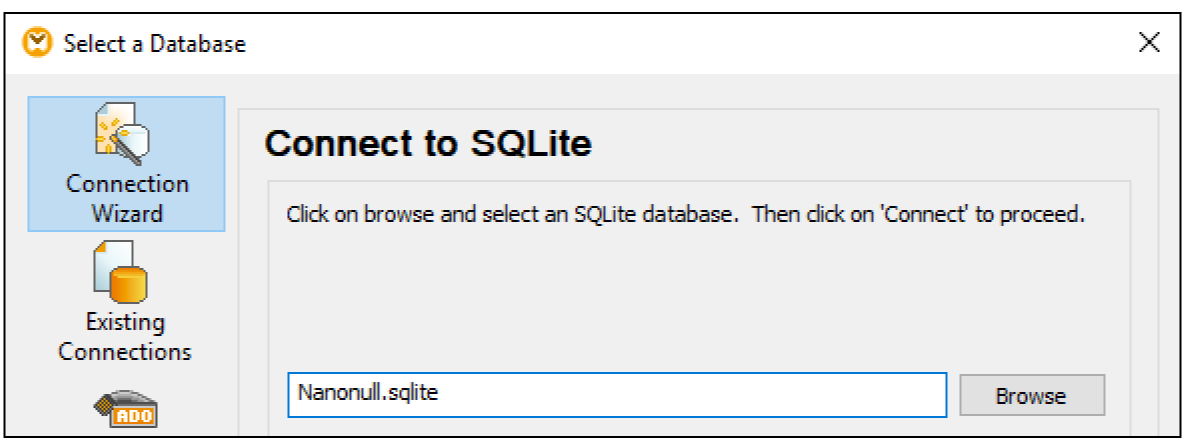
When the check box *Save all file paths relative to MFD file* is selected, MapForce will keep track of the files referenced by the component even when you save the mapping to a new folder. If all the files are in the same directory as the mapping, the path references will not be broken when you move the entire directory to a new location on the disk.

The setting *Save all file paths relative to MFD file* applies to the following files:

- Structural files used by complex input or output parameters of user-defined functions and variables of complex type;
- Input or output flat files (*Professional and Enterprise editions*);
- Schema files referenced by database components which support XML fields (*Professional and Enterprise editions*);
- Database files (*Professional and Enterprise editions*);
- Input or output XBRL, FlexText, EDI, Excel 2007+, JSON files (*Enterprise Edition only*).

Example: Database component (Professional and Enterprise editions)

When you add a database file such as Microsoft Access or SQLite to the mapping, you can enter a relative path instead of an absolute one in the **Select a Database** dialog box (*see screenshot below*). Before entering relative file paths, make sure to save the `.mfd` file first. If you want to change the path of a database component which is already in the mapping, click **Change** in the **Component Settings** dialog box.



Note: When you generate program code, compile MapForce Server execution files (**.mfx**), or deploy the mapping to FlowForce Server, a relative path will be converted to an absolute path if you select the check box *Make paths absolute in generated code* in the mapping settings 103. To find out more, see About Paths in Generated Code 76.

3.1.3.2 Paths in Execution Environments

If you generate code from mappings, the generated files are run by the target environment you have chosen: for example, RaptorXML Server. For the mapping to run successfully, any relative paths must be meaningful in the environment where the mapping runs. The base paths for each target language are given below:

Target language	Base path
XSLT, XSLT2, XSLT3	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 file (**.mfd**).
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*

Relative path to absolute path

When you generate program code, compile MapForce Server execution files (**.mfx**), or deploy the mapping to FlowForce Server, a relative path will be converted to an absolute path if you select the check box **Make paths absolute in generated code** in the mapping settings 103.

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

- Input and output instance files for all file-based components;
- Access and SQLite database files used as mapping components (*Professional and Enterprise editions*).

Library paths in generated code

Mapping files may optionally contain path references to different libraries. For example, you can import user-defined functions from another mapping file, from custom XSLT, XQuery*, C#* or Java* libraries, or from **.mff*** (MapForce Function) files. For more information, see Managing Function Libraries 194.

** Features available in MapForce Professional and Enterprise editions*

The option **Make paths absolute in generated code** applies only to mapping components and does not affect paths to external libraries. For all libraries other than XSLT and XQuery, the library path will be converted to an absolute path in generated code. For example, if your mapping file contains library references such as .NET **`.dll`** or Java **`.class`** files, and if you want to run the generated code in some other environment, the referenced libraries must exist at the same path in the target environment.

If you plan to generate an XSLT or XQuery file from a mapping, make the library path relative to the generated XSLT or XQuery file:

1. Open the mapping settings 103.
2. Select the check box **Reference libraries with paths relative to generated XSLT/XQuery file**. Make sure that the XSLT or XQuery library file exists at that path.

3.2 Connections

A connection is a line that connects a source to a target. Connections represent visually how data is mapped from one node to another. The subsections below describe different connection-related actions you can perform.

Create, copy, delete a connection

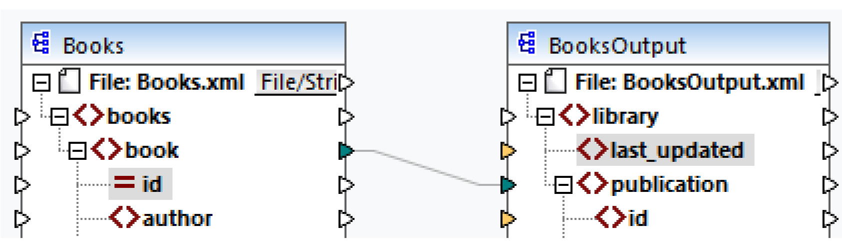
To create a connection between two items, press and hold the output connector[66] of a source node and drag it to a destination node. An input connector accepts *only one* connection. If you try to add a second connection to the same input, MapForce will ask if you want to replace the connection with a new one or duplicate the input item[72]. An output connector can have several connections, each to a different input.

To copy a connection to a different item, press and hold the thick section at the end of the connection (*see screenshot in Move a connection*) and drag it to the selected destination while holding the **Ctrl** key.

To delete a connection, click the connection and press the **Delete** key. Alternatively, right-click the connection and click **Delete** in the context menu.

Mandatory inputs

To help you in the mapping process, MapForce highlights mandatory inputs in orange in target components. The example below shows that as soon as you connect the `book` element of the `Books` component to the `publication` element of the `BookOutput` component, the connectors of the mandatory nodes of the `BooksOutput` component will be highlighted. If you do not connect mandatory inputs, the respective nodes will not be mapped to the target, and the mapping will be invalid.



Missing parent connections

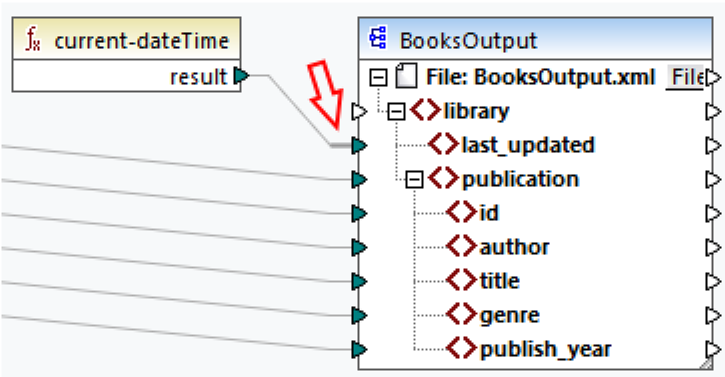
When you create connections between source and target node manually, MapForce analyzes possible mapping outcomes. If you connect two child nodes without connecting their parent nodes, you will see a notification message that suggests connecting the parent of the source node with the parent of the target node. This notification message helps you prevent situations where a single child node appears in the **Output** pane.

If you want to disable such notifications, take the following steps:

1. Go to the **Tools** menu and click **Options**.
2. Open the **Messages** group.
3. Clear the check box **When creating a connection, suggest connecting ancestor items**.

Move a connection

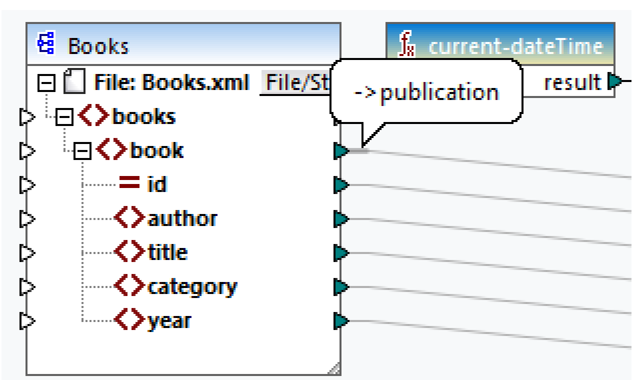
To move a connection to a different node, press and hold the thick section at the end of the connection (*see screenshot below*) and drag it to the selected destination.



See connection tooltips

Connection tooltips allow you to see the names of (i) nodes to which data is mapped or (ii) nodes from which data is mapped. To be able to see tips, press the toolbar button (**Show tips**). To see the names of nodes to which data is mapped, point the cursor at the thick section of a connection near the output connector (*see screenshot below*). To see the name of a node from which data is mapped, point the cursor at the thick section of a connection near the input connector. If multiple connections have been defined from the same output, maximum ten item names will be displayed in the tooltip.

In the example below, the target node to which the data from the `book` element is mapped is called `publication`.



Change connection settings

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

- Select a connection. Then go to the **Connection** menu and click **Properties**.
- Double-click the connection.
- Right-click the connection and click **Properties**.

For more information, see Connection Settings 87.

Highlight connections selectively

MapForce allows you to selectively highlight connections in a mapping. This feature may be useful when your mapping has many components with multiple connections. Highlighting connections selectively will make it easier to check whether the nodes of the selected component are mapped correctly. Note that the term *connector* used for the toolbar buttons below refers to a connection, i.e., a line connecting component nodes. See the available options below.

	**Show selected component connectors** (only direct connections)
	**Show connectors from source to target** (direct and indirect connections)

Only direct connections

When the **Direct-connections** button is *not* pressed, you can see all connections in black. When the **Direct-connections** button is pressed, only connections related to the currently selected component are black. Other connections are light gray.

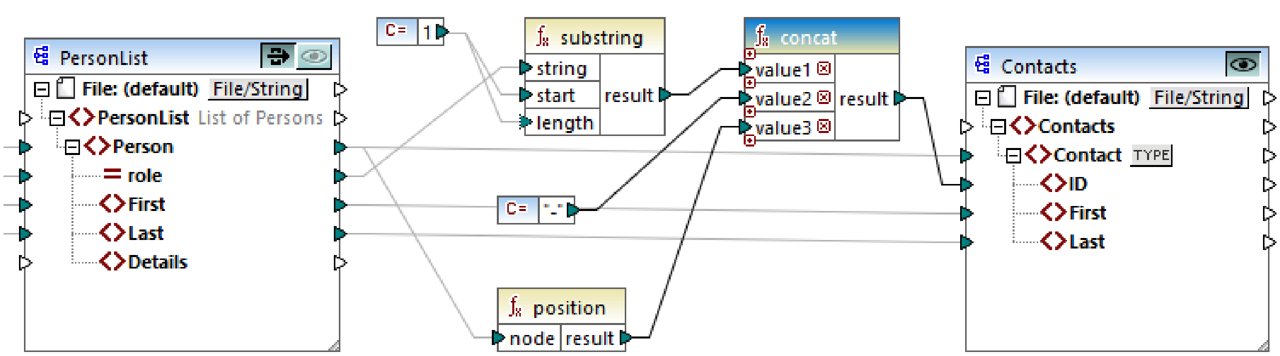
Direct and indirect connections

The **Source-to-target Connections** button becomes available only when the **Direct-connections** button is pressed. When the **Source-to-target Connections** button is pressed, you can trace connections of the currently selected component, including its direct connections and connections of its connected components up to the source and target files.

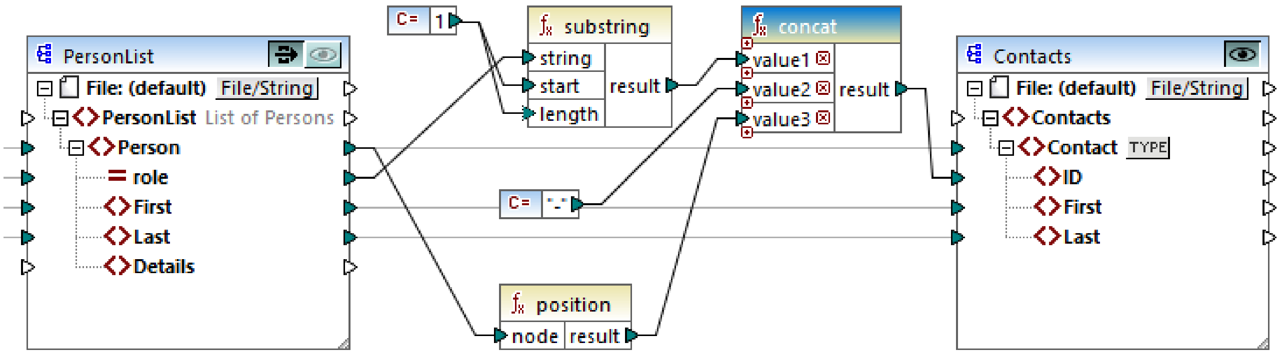
To understand how these two options work, see the example below.

Example

The screenshot below illustrates a part of the **`ChainedPersonList.mfd`** mapping, which is available in the **`MapForceExamples`** folder. In the mapping below, we have pressed the **Direct-connections** button, clicked the header of the **`concat`** component but have not yet pressed the **Source-to-target Connections** button. Therefore, we can now see that only the direct connections of the **`concat`** function to the **`"-"`** constant, the `substring`, `position`, and `Contacts` components are black. The other connections in the mapping are light gray.



The next step is to press the **Source-to-target Connections** button. The screenshot below reflects the changes:



With the **Source-to-target Connections** button pressed, some other connections have become black: (i) the connections of the **substring** function to the **1** constant and the `PersonList` component, and (ii) the connection of the **position** function to the `PersonList` component. However, the connections between the `PersonList` component and its preceding component remain light gray. Thus, when you press the **Source-to-target Connections** button and click on a component, you can trace the component's direct connections. If the selected component is connected to some transformation components [68] (e.g, functions, constants, filters, sort components, SQL-NoSQL-WHERE/ORDER components, if-else conditions, value-maps), you will also be able to see their connections up to the structural components [67] (such as `PersonList` above), variables, join components, or Web service functions, to which these transformation components are connected.

In this section

This section gives an overview of connections and is organized into the following topics:

- Connection Types 81
- Connection Settings 87
- Connection Context Menu 89
- Faulty Connections 90
- Keep Connections after Deleting Components 92

3.2.1 Connection Types

The following connection types are available in MapForce:

- Target-driven connections [81] (Standard)
- Source-driven connections [82] (Mixed Content)
- Matching-children connections [84]
- Copy-all connections [86] (Copy Child Items)

Target-driven vs. source-driven connections

Target-driven and source-driven connections are mutually exclusive. The choice between these two options depends on the order in which nodes need to be mapped. In target-driven connections, the order of nodes in the output is determined by the *target* schema. This connection type is suitable for most mapping scenarios and is the default connection type in MapForce. A target-driven connection is shown as a solid line (*see screenshot below*).

Target-driven connections might not be suitable when you want to map XML nodes with mixed context (child nodes and text). In this case, a source-driven connection [82] is recommended: The order of nodes in the output is determined by the *source* schema.

Matching-children and copy-all connections

Matching-children and copy-all connections belong to a subset of target-driven and source-driven connections. Matching-children and copy-all connections map data between nodes with child nodes that are similar or the same in the source and target components. Copy-all connections are similar to matching-children connections but have only one thick connection instead of multiple connections, which prevents the mapping area from being visually cluttered.

This section provides information about each connection type and the scenarios when these connection types are useful.

3.2.1.1 Source-Driven Connections

A source-driven connection enables you to automatically map mixed content (text and child nodes) in the same order as in the XML *source* file. A mixed-content connection is shown as a dotted line at parent-node level (*see Mapping the <para> element*). This topic explains how to map mixed content. It also shows the effect of using standard (target-driven) connections with mixed content.

Note: Source-driven connections can also be used in database fields with mixed-content (*Professional and Enterprise editions*).

Note: In order to accept mixed content, target components must have mixed-content nodes.

Mapping mixed content

This topic explains how to map mixed content using a source-driven connection. You will need the following files: **Tut-OrgChart.xml**, **Tut-Orgchart.mfd**, **Tut-Person.xsd**, and **Tut-OrgChart.xsd**, which are available in the Tutorial folder [15].

Source XML instance

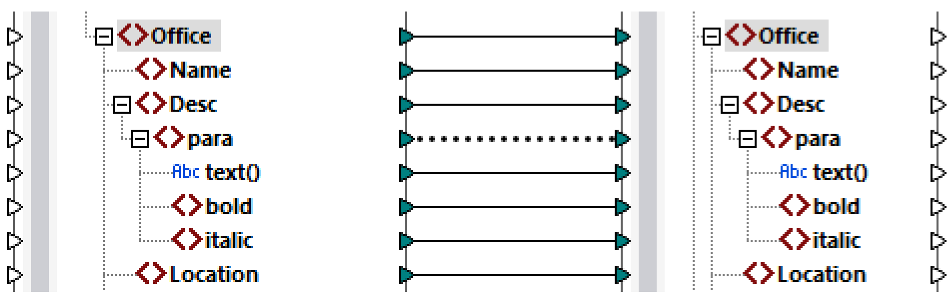
A snippet of **Tut-OrgChart.xml** is shown below. In this example, we will focus on the mixed-content element <para> with its child nodes <bold> and <italic>. The <para> element also contains a processing instruction (<?sort alpha-ascending?>) and a comment (<!--Company details... -->), both of which can also be mapped, as shown below. Note the sequence of the text and bold/italic nodes in the XML instance file.

```
<Desc>
  <para>The company was established in <bold>Vereno</bold> in 1995. Nanonull develops nanoelectronic technologies for <italic>multi-core processors.</italic> February 1999 saw the unveiling of the first prototype <bold>Nano-grid.</bold> The company hopes to expand its operations <italic>offshore</italic> 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.
  </para>
</Desc>
```

Mapping the <para> element

The image below illustrates a portion of **`Tut-Orgchart.mfd`**. In the example below, the dotted line shows that the `<para>` element has mixed content. To create mixed-content connections, take the following steps:

1. Select the menu command **Connection | Auto Connect Matching Children**, which will connect matching child nodes 84 automatically. Alternatively, you can manually map the `<para>` node with its child nodes.
2. Connect the `<para>` item in the source component with the `<para>` item in the target component. A message box will ask if you would like to define the connection as source-driven.
3. Click **Yes** to create a mixed-content connection.
4. Click the **Output** pane to see the result of the mapping. Click the button (**Wrap**) in the **Output** pane toolbar to view the full (i.e., not going beyond the scroll bar) code listing in the **Output** pane. The mixed content of the `<para>` node has been mapped in the same order it appears in the XML source file.



Processing instructions and comments

If your mapping has processing instructions and/or comments and you want to map them, take the steps below:

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

Target-driven connections with mixed content

Choosing target-driven connections for mixed content may have undesirable consequences. To see how target-driven connections affect the order of mixed-content nodes, follow the instructions below:

1. Open **`Tut-OrgChart.mfd`** from the **`Tutorial`** folder.

2. Press the toolbar button (Auto Connect Matching Children [84]). Clear the check box **Create copy-all connections** in the settings for matching-children connections [84]. This will prevent MapForce from creating copy-all connections [86] automatically.
3. Create a connection between the `para` node in the source and the `para` node in the target. A message will ask if you would like to define the connections as source-driven. Click **No**. This creates a target-driven connection.
4. Click the **Output** pane to see the result of the mapping (*screenshot below*).

```
<Desc>
    <para>The company was established in  in 1995. Nanonull develops nanoelectronic technologies for  February 1999 saw the unveiling of the first prototype  The company hopes to expand its operations  to drive down operational costs.
        <bold>Vereno</bold><bold>Nano-grid.</bold><italic>multi-core processors.</italic><italic>offshore</italic></para>
    <para>White papers and further information will be made available in the near future.
    </para>
</Desc>
```

The screenshot above shows that the content of the `text()` item in the source has been mapped to the target. However, the order of the child nodes (`bold` and `italic`) in the output corresponds to the order of these nodes in the target XML schema. This means that the `bold` and `italic` elements are not integrated into the text but are mapped separately.

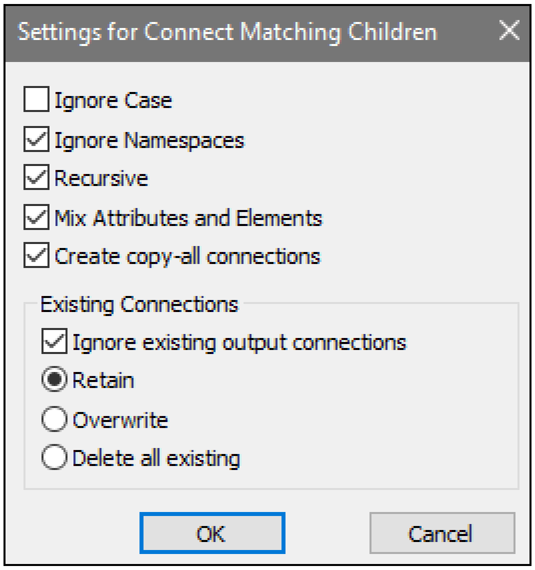
3.2.1.2 Matching-Children Connections

Matching-children connections automatically connect all child nodes which have the same names in the source and target files. To enable this option, do one of the following:

- Click the toolbar button (**Auto Connect Matching Children**).
- Go to the **Connection** menu and click **Auto Connect Matching Children**.

Settings for matching-children connections

To change the settings for matching-children connections, right-click any connection and select the option **Connect Matching Children** from the context menu (*see screenshot below*). Alternatively, go to the **Connection** menu and click **Settings for Connect Matching Children**.



The list below describes the options available in the dialog box **Settings for Connect Matching Children**. The settings in this dialog box apply only when the toolbar button (**Toggle auto connect of children**) is active.

General settings

- *Ignore Case:* Matching children will be connected regardless of the case of child node names.
- *Ignore Namespaces:* Matching children will be connected regardless of the namespaces of child nodes.
- *Recursive:* This option creates new connections between any matching nodes if they have the same names. It does not matter how deep the nodes are nested in the tree.
- *Mix Attributes and Elements:* This option allows creating connections between attributes and elements that have the same names. For example, a connection is created if two `Name` nodes exist, even though one is an element, and the other is an attribute.
- *Create copy-all connections:* This setting is active by default. It creates (if possible) a copy-all connection 86, which maps data between nodes with child nodes that are similar or the same.

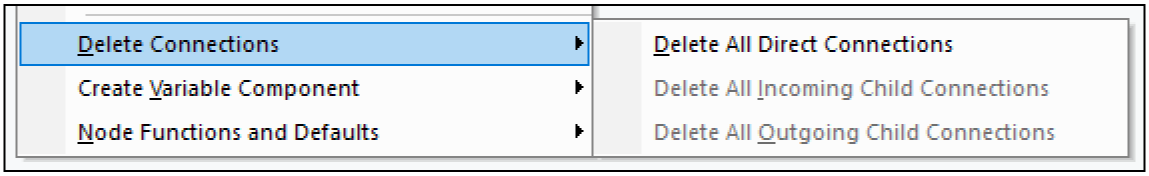
Existing connections

- *Ignore existing output connections:* This option creates additional connections for any matching nodes even if they already have outgoing connections.
- *Retain:* This option keeps existing connections.
- *Overwrite:* This option overwrites existing connections.
- *Delete all existing:* This option deletes all existing connections before creating new ones.

Delete connections as a group

If you want to delete connections as a group, follow the instructions below:

1. Right-click a node name in the component.
2. Select **Delete Connections | Delete All <...> Connections** from the context menu (*see screenshot below*).



- *Delete All Direct Connections:* This option deletes all connections that are directly mapped to or from the selected node.
- *Delete All Incoming Child Connections:* This option is active only if you have right-clicked a parent node in a target component. This option deletes all incoming child connections of the selected parent node.
- *Delete All Outgoing Child Connections:* This option is active only if you have right-clicked a parent node in a source component. This option deletes all outgoing child connections of the selected parent node.

3.2.1.3 Copy-All Connections

Copy-all connections map data between nodes with child nodes that are similar or the same. Copy-all connections are possible only for identical formats (e.g., JSON to JSON or XML to XML). This principle also applies to all text components: flat files, FlexText and EDI files. Since these formats are all text files, you can combine any of them and create a copy-all connection between EDI and FlexText files, for example. See Mapping: Sources and Targets 15 to find out more about the formats that can be used as data sources and targets.

The main benefit of copy-all connections is that they visually simplify the mapping workspace: One connection, represented by a thick line, is created instead of multiple connections (*see example in Create Copy-all Connections Manually*). The subsections below explain how to create copy-all connections automatically and manually.

Create copy-all connections automatically

To create a copy-all connection automatically, take the following steps:

1. Go to the **Connection** menu.
2. Click **Settings for Connect Matching Children**.
3. Check the box **Create copy-all connections** and click **OK**.
4. Press the toolbar button **Toggle auto connect of children**. Alternatively, go to the **Connection** menu and click **Auto Connect Matching Children**.

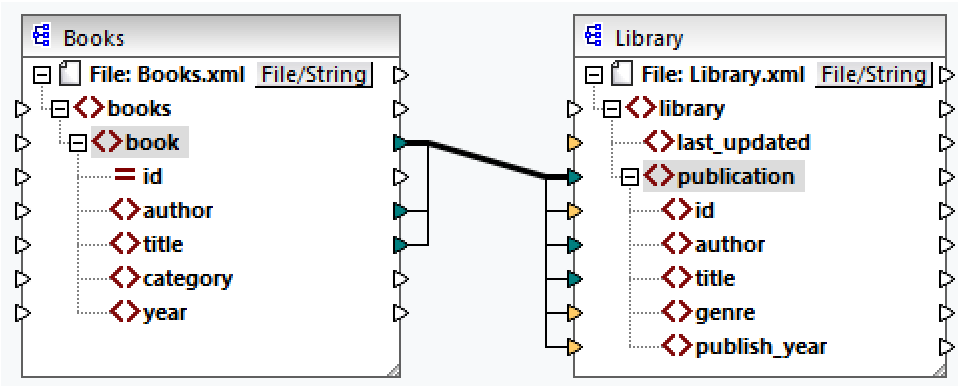
If types and/or names of child nodes in the source and target are not the same, a copy-all connection will not be created automatically, and you will need to create it manually.

Create copy-all connections manually

To create a copy-all connection manually, take the following steps:

1. Add a source file: Click **XML Schema/File** in the **Insert** menu and browse for **`Books.xml`** located in the BasicTutorials folder 15.
2. Add a target file: Click **XML Schema/File** in the **Insert** menu and browse for **`Library.xsd`** located in the same folder as **`Books.xml`**. Click **Skip** when MapForce suggests adding an XML sample file.

3. Map the <book> node of the **`Books`** component to the <publication> node of the **`Library`** component. As the structures of the <book> and <publication> elements do not fully coincide, the copy-all connection is not created. Instead, the **Auto Connect Matching Children** function automatically connects all the child nodes with the same name, which is discussed in Tutorial 1[37].
4. To change the automatic connection to a copy-all connection, right-click the connection between <book> and <publication> and select **Copy-All (Copy Child Items)** from the context menu.
5. A pop-up window will suggest replacing the existing connections with a copy-all connection. Click **OK**. Now the source and target have a copy-all connection (*see screenshot below*).



In the mapping above, only two child nodes are identical in the two structures: <author> and <title>. Therefore, a copy-all connection exists between these nodes. Child nodes that are not the same cannot be connected. The screenshot shows that `id` is not included in the copy-all connection, because its type is not the same in the source and target: `id` is an attribute in the source and an element in the target. If you try to create a connection between nodes that are not the same, e.g., <category> and <genre>, MapForce prompts you to replace this connection or duplicate the input.

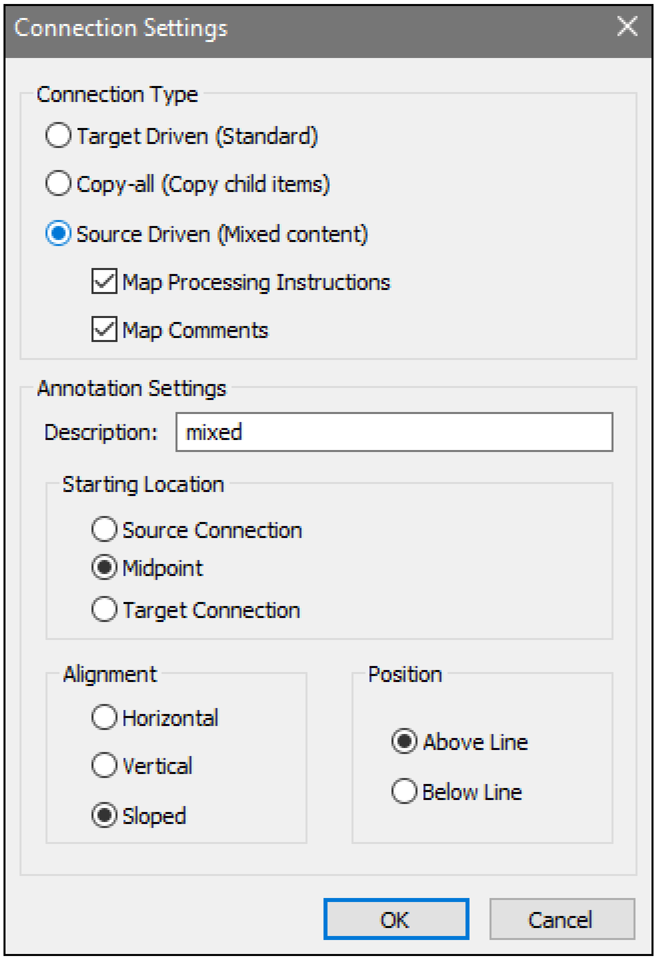
Duplicating input[72] only makes sense if you want the target to accept data from more than one input, which is not required here. If you choose to replace the copy-all connection, a message box prompts you again to resolve or delete the copy-all connection. Click **Resolve copy-all connection** if you want to replace the copy-all connection with individual target-driven connections[81]. If you prefer to remove the copy-all connection completely, click **Delete child connections**.

Important:

When you create a copy-all connection between a schema and a parameter of a user-defined function[198], the two components must be based on the same schema. It is not necessary that they both have the same root elements, however.

3.2.2 Connection Settings

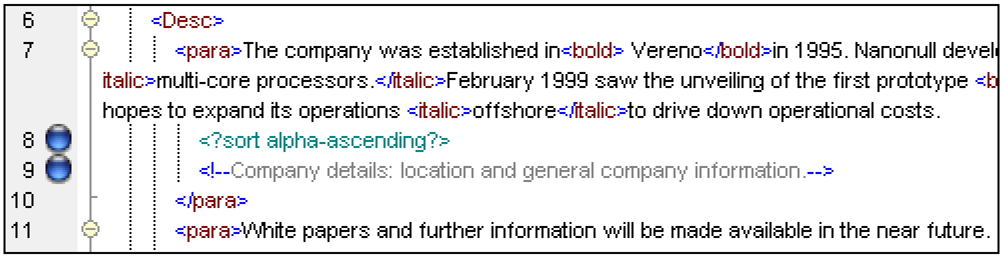
The **Connection Settings** dialog box defines the settings of a connection. To open this dialog box, double-click the connection. Alternatively, right-click a connection and select **Properties** from the context menu. The settings are divided into two parts: connection types and annotation settings. For more information, see the subsections below.



⊟ Connection types

You can choose one of the connection types described below:

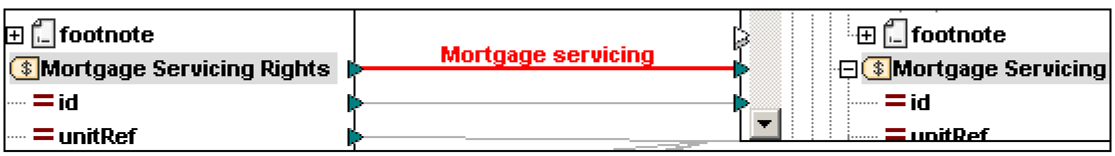
- Target-driven (Standard) [81] connections are suitable for most mapping scenarios.
- Copy-all (Copy child items) [86] connections: If a source and target components have identical or similar nodes with matching child nodes, a copy-all connection will automatically be created between these matching nodes.
- Source-driven (mixed content) [82] connections map mixed content (text and child nodes) in the same order as in the XML source file. If you select **Map Processing Instructions** and/or **Map Comments**, you will be able to include these data groups in the output file (*see screenshot below*).



Annotation settings

The **Annotation Settings** section enables you to label a connection. This option is available for all connection types. To annotate a connection, follow the instructions below:

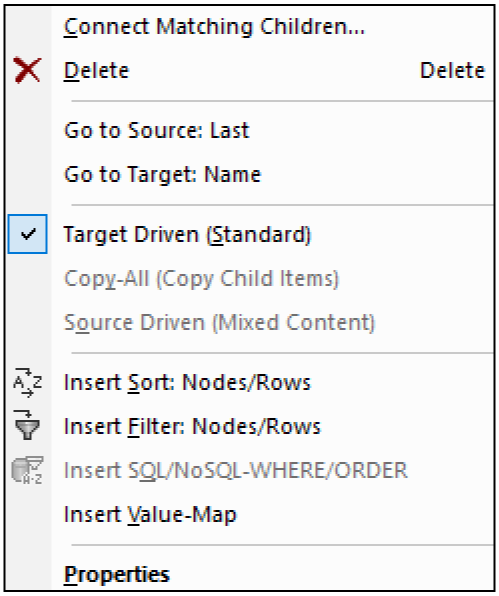
1. Right-click the connection and select **Properties** from the context menu. Alternatively, double-click the connection.
2. Enter the name of the selected connection in the **Description** field. This enables all the options in the **Annotation Settings** section.
2. Use the remaining groups to define the **Starting Location**, **Alignment** and **Position** settings of the label.
3. Press the toolbar button (**Show Annotations**). If the button is not yet visible in the toolbar, activate the button in the **View Options** toolbar.



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

3.2.3 Connection Context Menu

This topic describes commands available in the connection context menu. When you right-click a connection, the following context commands become available:



For more information, see the subsections below.

General settings

- *Connect Matching Children:* Opens the Connect Matching Children [84] dialog box. This command is enabled when the connection is allowed to have matching children.
- *Delete:* Deletes the selected connection.
- *Go to Source: <item name>:* Highlights the output connector [66] of the selected connection.
- *Go to Target: <item name>:* Highlights the input connector [66] of the selected connection.

Connection types

See details about connection types in Connection Types [81] and Connection Settings [87].

Insert commands

- *Insert Sort: Nodes/Rows:* Adds a sort [162] component between a source node and a target node.
- *Insert Filter: Nodes/Rows:* Adds a filter [168] component between a source node and a target node.
- *Insert SQL/NoSQL-WHERE/ORDER:* Adds an SQL/NoSQL-WHERE/ORDER component between a source node and a target node (*Professional and Enterprise editions*).
- *Insert Value-Map:* Adds a value-map [174] between a source node and a target node.

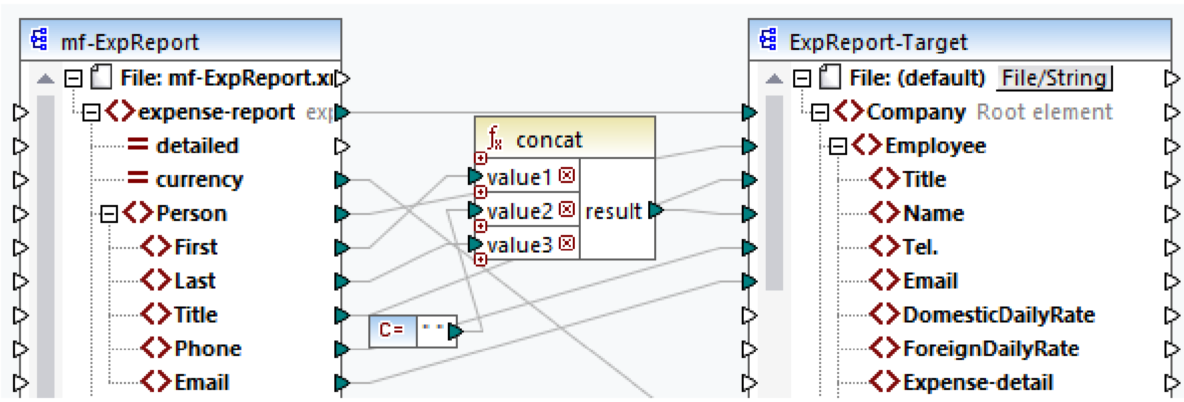
Properties

Opens the Connection Settings [87] dialog box.

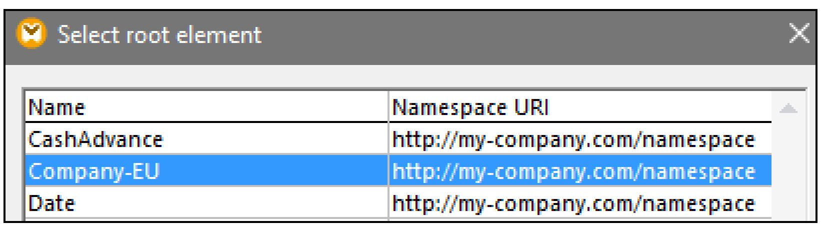
3.2.4 Faulty Connections

There are situations in which you might want to change a schema of a source or target. Changes to a schema can affect the validity of your mapping and result in several faulty connections. This topic explains how to fix such connections after you have changed the schema file. Follow the instructions in the example below to understand how to deal with faulty connections.

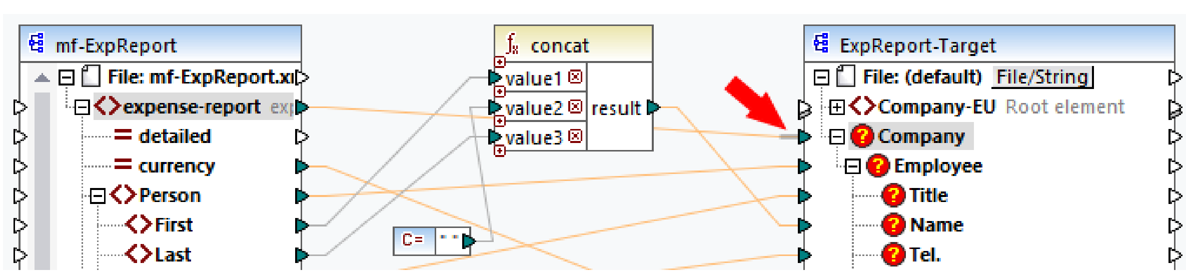
1. Open **`Tut-ExpReport.mfd`** available in the Tutorial folder[15]. The portion of this mapping is shown below.



2. Open **`ExpReport-Target.xsd`** in an editor (e.g., Altova XMLSpy) and change the `Company` root element in the target schema to `Company-EU`. You do not need to close MapForce.
3. After you have edited the root element of the target schema**,** the **Changed files** prompt appears in MapForce. Click the **Reload** button. Since the root element has been changed, the component displays multiple faulty nodes.
4. Click **Select new root element** at the top of the component (*see screenshot below*). You can also change the root element by right-clicking the component header and selecting **Change Root Element** from the context menu.



5. Select `Company-EU` as the new root element and click **OK**. The `Company-EU` root element is now visible at the top of the component.
6. Now you need to move the connection from the faulty `Company` node to the new root element. Press and hold the thick section (*see red arrow below*) of `Company`'s connection. Then drag the connection to the `Company-EU` root element.



A notification dialog box will ask whether you would like to move all the matching connected child nodes. You can choose between moving only the selected connection or the selected connection with its child nodes that match the child nodes in the new root element. In our example, we have chosen the option **Include descendent connections**. As soon as you click this button, all the faulty nodes will disappear from the component.

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

Alternative solution

An alternative solution to the problem discussed above could be deleting the faulty nodes you may no longer need in your mapping. For example, when you delete the connection between the **`concat`** function and `Name`, the `Name` node will disappear from the `ExpReport-Target` component.

Faulty connections in databases (Professional and Enterprise editions)

If your database component has faulty connections, you will need to change the component settings 71. Clicking the **Change** button in the **Component Settings** dialog box allows you to select a different database or change tables in your database component. All valid/correct connections and relevant database data will be kept if you select a database with the same structure.

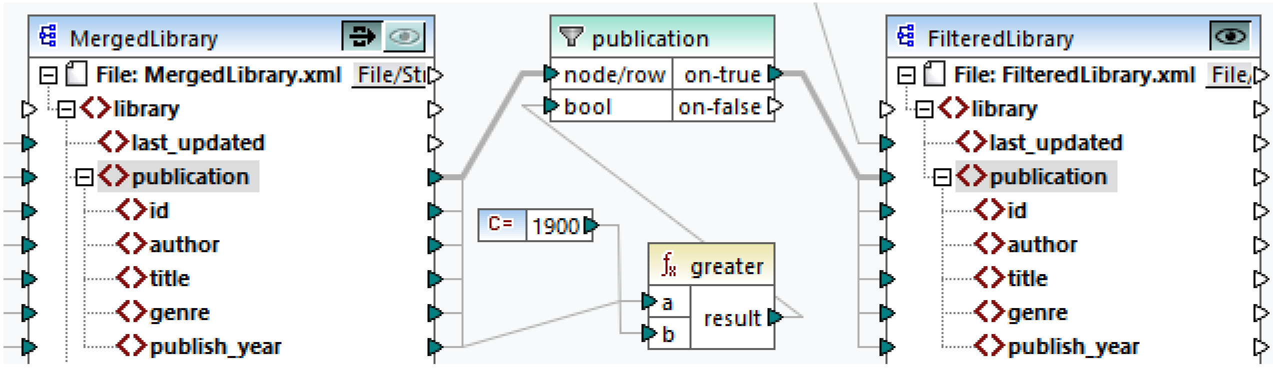
3.2.5 Keep Connections after Deleting Components

MapForce allows you to keep connections even after deleting some transformation components 67: e.g., variables, sort and filter components, value-maps, simple inputs, SQL/NoSQL-WHERE/ORDER components. Connections can be single or multiple. Keeping connections might be particularly useful with multiple child connections, because you will not have to restore every single child connection manually after deleting a transformation component. To enable this option, go to **Tools | Options | Editing** and select **Smart component deletion (keep useful connections)**. By default, this option is disabled, which means that deleting a transformation component will also delete its direct connections.

Example

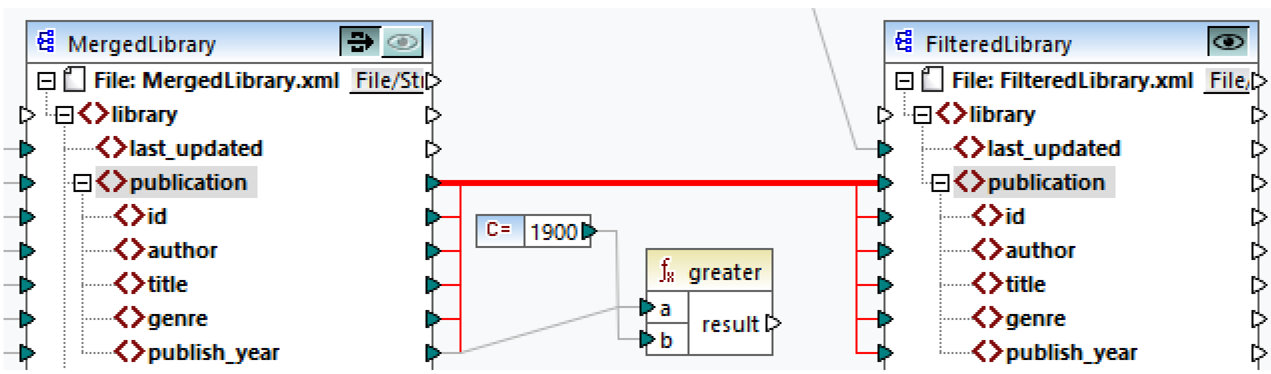
The sample file called **`Tut3-ChainedMapping`** is used to illustrate smart component deletion. The sample file is available in the BasicTutorials 15 folder.

Before deletion
The screenshot below shows that copy-all connections 86 exist between the `MergedLibrary` component and the `publication` filter, and between the `publication` filter and the `FilteredLibrary` component. Now we want to delete the `publication` filter but keep the copy-all connections. In order to do that, select the check box **Smart component deletion** in the **Options** dialog box (*see above*).



After deletion

After the `publication` function has been deleted, the copy-all connection has been created directly between the `publication` node in `MergedLibrary` and the `publication` node in `FilteredLibrary` (*see screenshot below*).



Note: If a filter component has both `on-true` and `on-false` outputs connected, the connections of both outputs will be kept.

3.3 General Procedures and Features

In addition to creating mappings, you can also validate your mapping and output, generate code, use text view features and define mapping settings. This section is organized into the following topics:

- Validation 94
- Code Generation 96
- Text View Features 96
- Text View Search 100
- Mapping Settings 103

3.3.1 Validation

This topic explains how to validate mappings. The topic also shows how to preview, save and validate your output.

Validate mappings

MapForce validates mappings automatically when you click the **Output** pane. You can also validate your mapping manually, which can help you identify and correct potential errors and warnings before running the mapping. To validate a mapping manually, click the **Mapping** pane and then do one of the following:

- Click **Validate Mapping** in the **File** menu.
- Click  (**Validate**) in the toolbar.

When you validate a mapping, MapForce checks, for example, for unsupported component types, incorrect or missing connections. To find out more about validation statuses, see Messages Window 26. The **Messages** window also allows you to take message-related actions 27. 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.

Validation of transformation components

Validation of transformation components 68 works as follows:

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

Preview and validate output

MapForce allows you to preview the 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 in MapForce before processing the generated code externally. When you preview the mapping results, MapForce executes the mapping and shows the output in the Output pane 29.

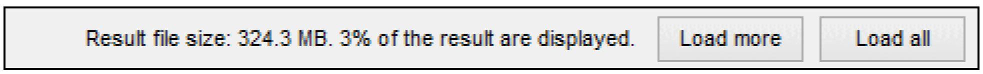
Once the data is available in the **Output** pane, you can validate and save it if necessary. You can also use the **Find** command (**Ctrl + F**) to quickly locate a particular text pattern in the output file. For more information, see Text View Search [100]. Any error, warning or information messages related to the mapping execution are displayed in the Messages window [26].

If you select C++, C#, or Java (*Professional and Enterprise editions*) as a transformation language [16], MapForce executes the mapping using its built-in transformation engine and displays the result in the **Output** pane.

To save the transformation output, click the **Output** pane and then do one of the following:

- Click **Save Output File** in the **Output** menu.
- Click (**Save generated output**) in the toolbar.

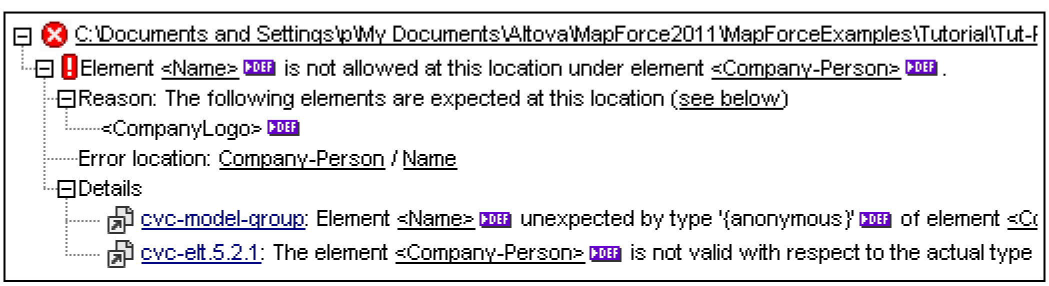
Load options
When you preview large output files, MapForce limits the amount of data displayed in the **Output** pane. In this case, the **Load more** button appears in the lower area of the pane (*see screenshot below*). Clicking the **Load more** button adds the next piece of data. You can configure the preview settings from the **General** tab of the **Options** dialog box. For more information, see MapForce Options [467].



Validate output
As soon as the output becomes available in the **Output** pane, you can validate the output against the schema associated with it. Note that 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. To validate the output, do one of the following:

- Open the **Output** pane and click (**Validate Output**) in the toolbar.
- Open the **Output** pane and click **Validate Output File** in the **Output** menu.

The screenshot below illustrates unsuccessful validation. The **Messages** window contains detailed information on the errors. For example, if you click the `<Name>` link, MapForce will highlight this element in the **Output** pane.



3.3.2 Code Generation

You can generate code from a mapping depending on the language you have selected as a data transformation language 16. You can generate code in the following languages:

- XSLT 1.0/XSLT 2.0/XSLT 3.0
- XQuery (*Professional and Enterprise editions*)
- Java (*Professional and Enterprise editions*)
- C# (*Professional and Enterprise editions*)
- C++ (*Professional and Enterprise editions*)

Example
If you want to generate XSLT code, follow the instructions below:

1. Select the menu item **File | Generate Code in | XSLT**.
2. Select the folder where 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 26.

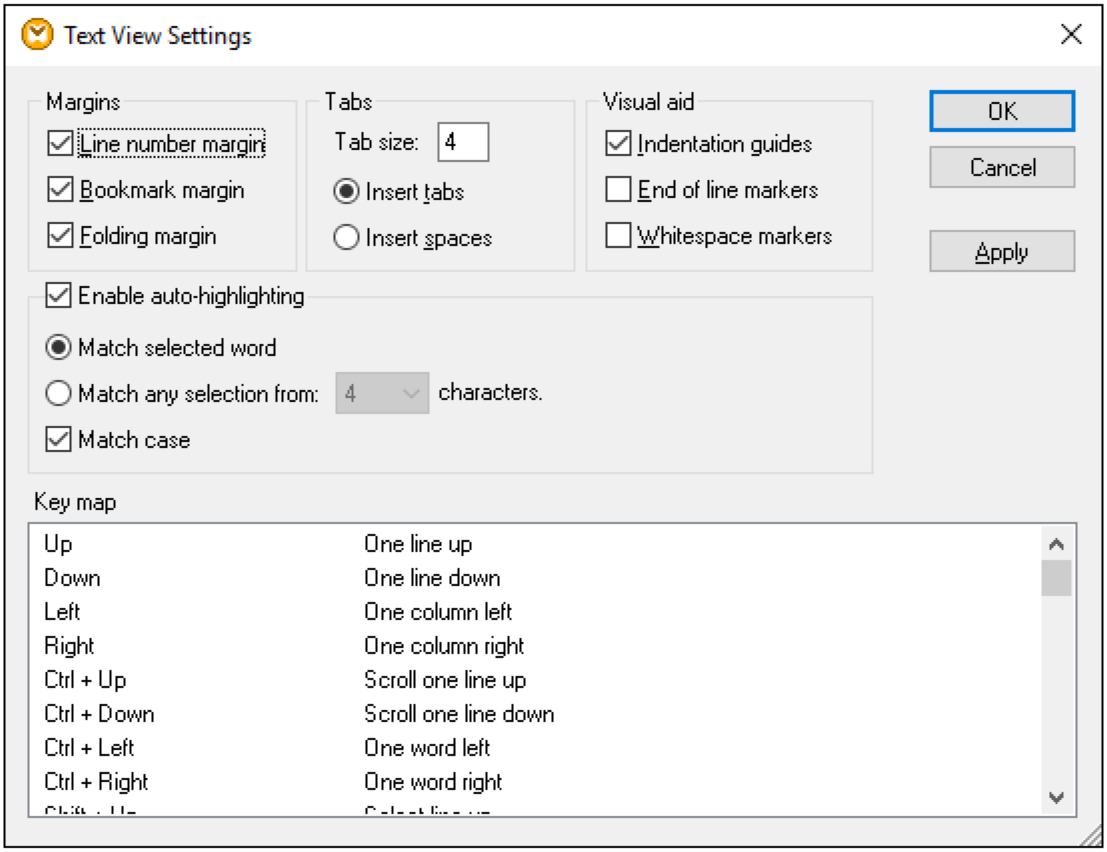
To preview the generated XSLT code, click the **XSLT** pane at the bottom of the **Mapping** window. The same steps apply to any of the languages listed above.

After the code generation has been completed, the destination folder will include the following two files:

1. An XSLT transformation file, named after the target schema. This transformation file has the following format: **`<Mapping>MapTo<TargetFileName>.xslt`**. `<Mapping>` is the value of the **Application Name** field in the mapping settings 103. `<TargetFileName>` 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. For more information, see Change Component Settings 71 and Library paths in generated code 76.
2. A **`DoTransform.bat`** file, which enables you to run the XSLT transformation with Altova RaptorXML Server from the command line. In order to run the command, you will need to install RaptorXML.

3.3.3 Text View Features

The Output pane 29 and the XSLT pane 28 have multiple visual aids to make the display of text easier: e.g., margins, text highlighting, indentation guides, end-of-line and whitespace markers. You can customize these features in the **Text View Settings** dialog box (*see screenshot below*). The settings in this dialog box apply to the entire application.



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

- Select **Output | Text View Settings**.
- Click  (**Text View Settings**) in the toolbar.
- Right-click the blank area in the **Output** pane and select **Text View Settings** from the context menu.

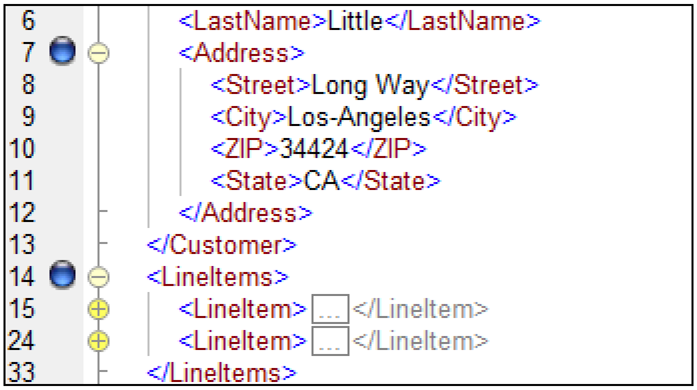
Some of the navigation aids can also be toggled from the **Text View** toolbar, the application menu, or keyboard shortcuts. For more more information about shortcuts, see the **Key map** section of the **Text View Settings** dialog box shown above.

See the list of available settings below.

⊟ Margins

Line number margin
Line numbers are displayed in the line number 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.

Bookmark margin
Lines in the document can be bookmarked for quick reference and access. If the **Bookmark margin** check box in the **Text View Settings** dialog box is selected, bookmarks are displayed in the bookmarks margin (*see screenshot below*). If the **Bookmark margin** check box is not selected, the bookmarked lines are highlighted in cyan.

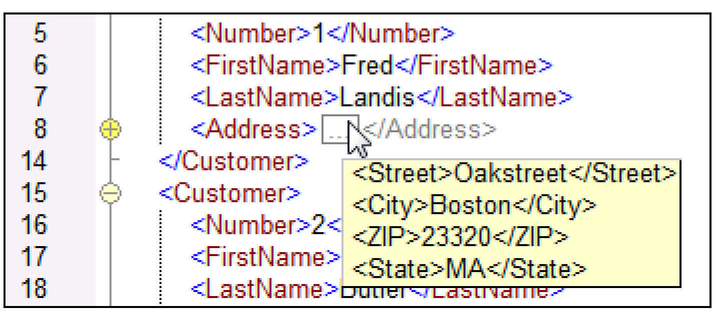


You can edit and navigate bookmarks using the commands given in the table below. The commands are available in the **Output** menu and also through the context menu when you right-click the **Output**, **XSLT** or **XQuery** pane.

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

Folding margin

Source folding refers to the ability to expand and collapse nodes. This feature is displayed in the source folding margin. The margin can be activated or disabled 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 (*see screenshot below*). To preview the collapsed code without expanding it, hover over the ellipsis. This opens a tooltip that displays the previewed code, as shown in the screenshot 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.



Enable auto-highlighting

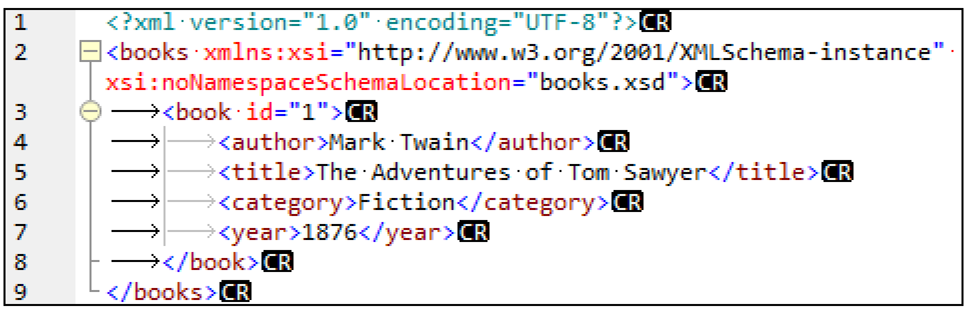
The **Enable auto-highlighting** setting allows you to see all the matches of the selected piece of text. The selection is highlighted in pale blue, and the matches are highlighted in light brown. The selection and its matches are indicated as gray marker-squares in the scroll bar. The current cursor position is shown as the blue cursor-marker in the scroll bar. A selection can be an entire word or a fixed number of characters. You can also specify whether case should be taken into account or not.

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

Visual aid

Indentation guides
Indentation guides are vertical lines that indicate the extent of a line's indentation. They can be toggled on and off in the **Text View Settings** dialog box. The **Insert tabs** and **Insert spaces** options take effect when you use the option **Output | Pretty-Print XML Text**.

End-of-line and whitespace markers
End-of-line and whitespace markers (*see screenshot below*) can be toggled on in the **Text View Settings** dialog box. The arrows represent tab characters. The *CR* abbreviation stands for a carriage return. The dots represent space characters.



Other text view settings

Syntax coloring
Syntax coloring is another visual aid that makes code listings more reader-friendly. Syntax coloring depends on 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.

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 separated from its parent by four space characters. This can be customized in the **Text View Settings** dialog box. To pretty-print an XML document, select the menu command **Output | Pretty-Print XML Text** or click (**Pretty-print**) in the toolbar.

Word wrapping

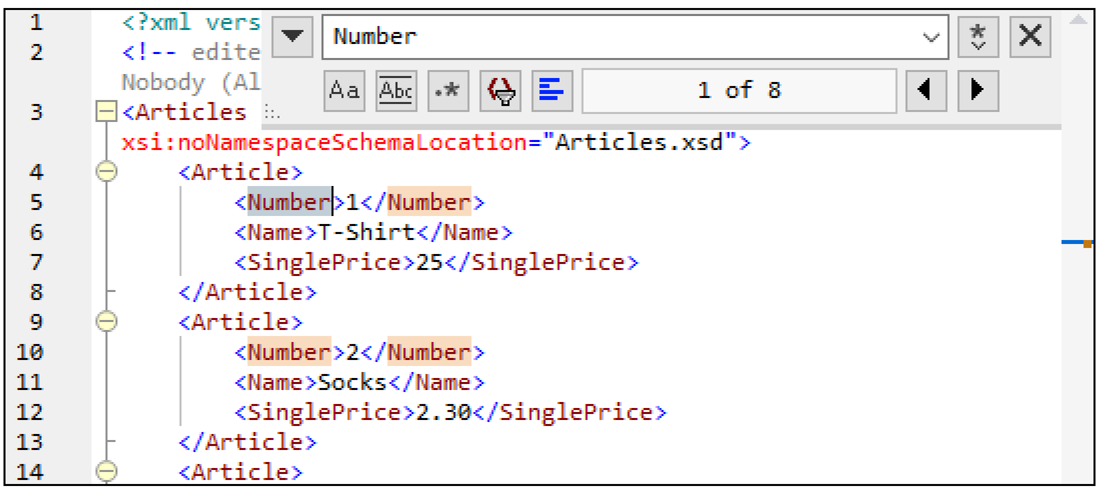
Word wrapping helps display a code listing within the borders of the working area. If the word wrap setting is not enabled, some portions of text may not be fully visible in the working area. To toggle word wrapping in the currently active document, select the menu command **Output | Word Wrap** or click (**Word Wrap**) in the toolbar.

3.3.4 Text View Search

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

You can search for a term in the entire document or within a text selection. To start a search , press **Ctrl+F** or select the menu command **Edit | Find**. You can enter a string or use the combo box to select a string from one of the last 10 strings. When you enter or select a string, all matches are highlighted, and the positions of the matches are indicated by orange markers in the scroll bar (*see screenshot below*). The position of the currently selected match (highlighted in gray) depends on where the cursor was last located.

You can see the total number of matches and the index position of the currently selected match. Use the (**Previous**) and (**Next**) buttons to switch between the matches.



Find options

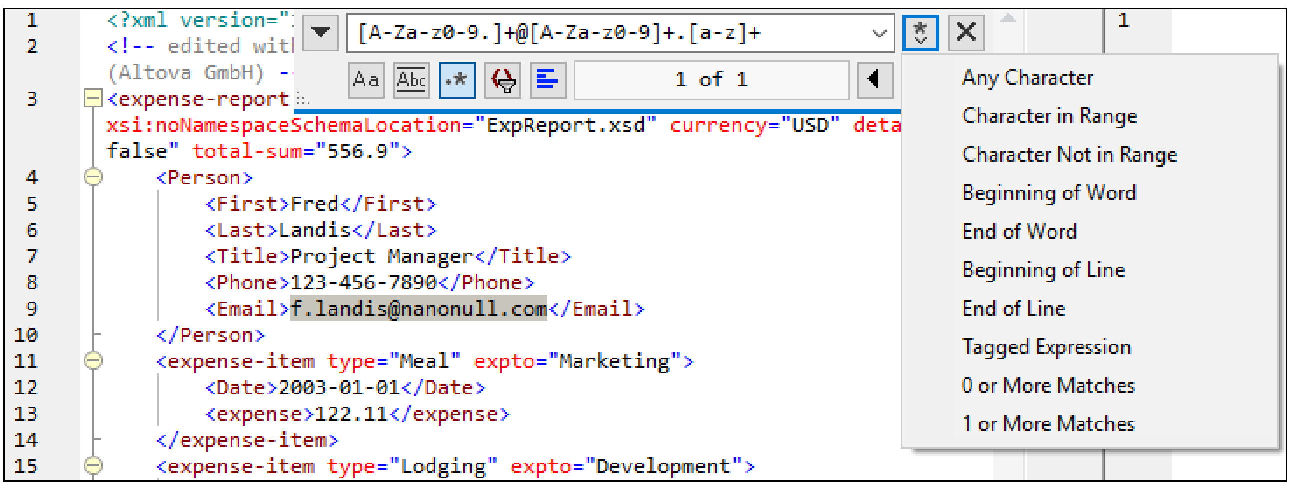
You can specify find criteria with the help of the buttons located under the search field. The list of the available options is given in the table below.

Option	Icon	Description
Match case		Does a case-sensitive search: e.g., *Address* is not the same as *address*.

Option	Icon	Description
Match whole word		Only identical words will match.
Use regular expression		If this option is toggled on, the search term will be read as a regular expression. See *Regular expressions* below.
Find anchor		The position of an anchor depends on the place where the cursor was last located. Clicking the **Previous** and **Next** buttons does not change the position of the anchor.
Find in selection		A selection is a marked piece of text. To find a term within a selection, mark a piece of text, press **Ctrl+F**, make sure the **Find in selection** button is pressed, and type the term in the search field.

Regular expressions

You can use regular expressions to find a text string. To do this, switch on the **Use regular expressions** option (*see table above*). Then enter a regular expression in the search field. Clicking (**Regular Expression Builder**) gives you a list of sample regular expressions (*see below*). The screenshot below shows a regular expression that helps find email addresses.



Regular expression metacharacters

The table below shows metacharacters that you can use to find and replace text. All the metacharacters except for the last two correspond to the menu items in **Regular Expression Builder** (*see above*).

Menu item	Metacharacters	Description
Any Character	`.`	Matches any character. This is a placeholder for a single character.
Character in Range	`[...]`	Matches any characters in this set. For example, **`[abc]`** matches any of the characters `a`, `b` or `c`. You can also use ranges: e.g., **`[a-z]`** for any lower case character.
Character Not in	`[^...]`	Matches any characters not in this set. For example, **`[^A-Za-z]`**

Menu item	Metacharacters	Description
Range		matches any character except an alphabetic character.
Beginning of Word	`\<`	Matches the beginning of a word.
End of Word	`\>`	Matches the end of a word.
Beginning of Line	`^`	Matches the beginning of a line unless it is used inside a set (*see above*).
End of Line	`$`	Matches the end of a line. For example, **`A+$`** matches one or more `A`'s at the end of a line.
Tagged Expression	`(abc)`	The parentheses mark the start and end of a tagged expression. Tagged expressions may be useful when you need to tag ("remember") a matched region to refer to it later. Up to nine sub-expressions can be tagged and then back-referenced later. For example, **`(the) \1`** matches the string `the the`. This expression can be explained as follows: Match the string `the` and remember it as a tagged region; the expression must be followed by a space character and a back-reference to the tagged region matched previously.
0 or More Matches	`*`	Matches zero or more matches of the preceding expression. For example, **`Sa*m`** matches `Sm`, `Sam`, `Saam`, `Saaam` and so on.
1 or More Matches	`+`	Matches one or more occurrences of the preceding expression. For example, **`Sa+m`** matches `Sam`, `Saam`, `Saaam` and so on.
	`\n`	Where `n` is `1` through `9`, `n` refers to the first through ninth tagged region (*see above*).
	`\x`	Allows using a character `x`, which would otherwise have a special meaning. For example, `\[` would be interpreted as `[` and not as the start of a character set.

Find special characters

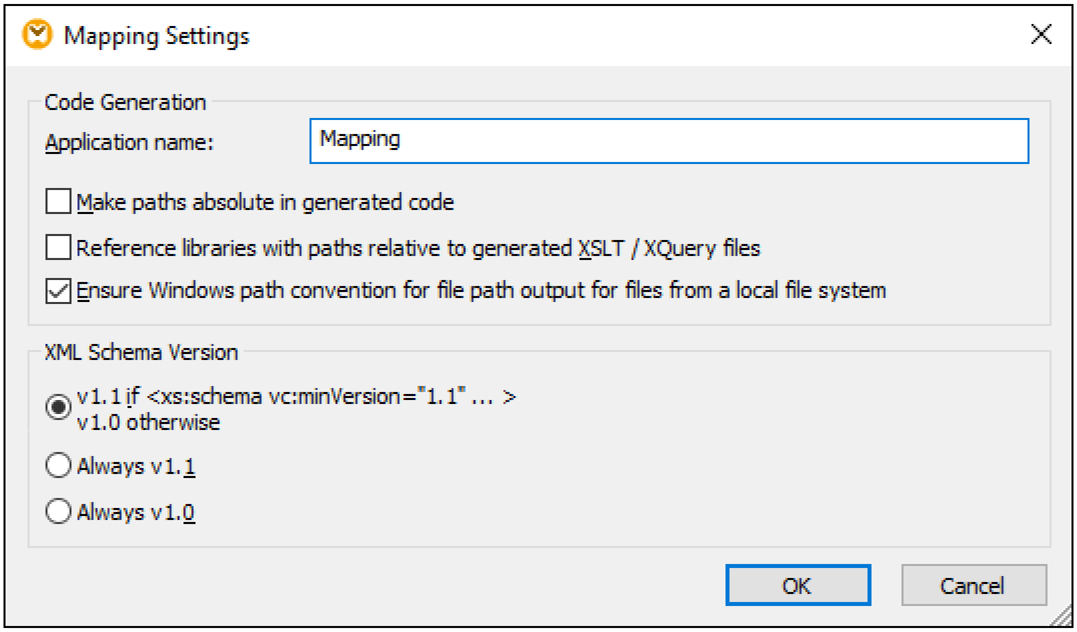
If the **Use regular expressions** option is enabled, you can search for any of the following special characters within the text:

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

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

3.3.5 Mapping Settings

The **Mapping Settings** dialog box (*see screenshot below*) allows you to define document-specific settings. To open this dialog box, go to the **File** menu and click **Mapping Settings**. Alternatively, right-click the empty area in the mapping pane and select **Mapping Settings** from the context menu.



The available settings are described in the subtopics below.

Code Generation

- *Application name:* Defines the prefix of the generated XSLT file or the name of the generated Java, C#, or C++ application (*Professional and Enterprise editions*).

- *Java base package name (Professional and Enterprise editions):* This option applies when Java is selected as a transformation language. The option defines the base package name of the Java output.

- Make paths absolute in generated code: This check box affects all paths in mapping components, except paths to external library files (e.g., XSLT libraries). The check box defines whether the file paths should be relative or absolute in the generated program code. For more information, see Paths in Execution Environments [76].

- *Reference libraries with paths relative to the generated XSLT/XQuery files:* This check box is applicable when the mapping language is XQuery (*Professional and Enterprise editions*) or XSLT. This option is useful if your mapping references an XSLT or XQuery library, and you plan to generate XSLT or XQuery files from the mapping. Select this check box if you want the library paths to be relative to the directory of the generated XSLT or XQuery code. If the check box is not selected, the library paths will be absolute in the generated code. See also Library paths in generated code [76].

- *Ensure Windows path convention for file path:* This check box is applicable when the mapping language is XQuery (*MapForce Professional and Enterprise editions*), XSLT 2.0 or XSLT 3.0. The check box makes sure that Windows path conventions are followed. When you output XSLT 2.0, XSLT 3.0 or XQuery, the currently processed file name is internally retrieved with the help of the **`document-uri`** function, which returns a path in the `file://URI` format for local files. When this check box is selected, a `file://URI` path specification is automatically converted to a complete Windows file path (e.g., `C:\...`) to simplify further processing.

Output File Settings (Professional and Enterprise editions)

The **Line ends** combo box allows you to specify the line endings of the output files. *Platform default* means the default option for the target operating system: e.g., Windows (CR+LF), macOS (LF), or Linux (LF). You can also select a specific line ending manually. The settings you select here are important when you compile a mapping to a MapForce Server Execution file (`.mfx`) or when you deploy a mapping to FlowForce Server running on a different operating system.

XML Schema Version

This option allows you to define the XML schema version used in the mapping file. Note that not all version 1.1 specific features are currently supported. If the `xs:schema vc:minVersion="1.1"` declaration is present, 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`, XSD 1.0 will be the default mode. Do not confuse the `vc:minVersion` attribute with the `xsd:version` attribute. The first attribute has the XSD version number, while the second attribute has the document version number. Changing this setting in an existing mapping causes the reloading of all schemas of the selected XML schema version and might also change its validity.

Web Service Operation Settings (Enterprise editions)

The **WSDL Definitions**, **Service**, **Endpoint** and **Operation** fields are automatically filled if the mapping document is part of a Web service implementation.

4 Structural Components

This section provides information about various data formats that you can use as data sources and targets:

- XML and XML Schema 106

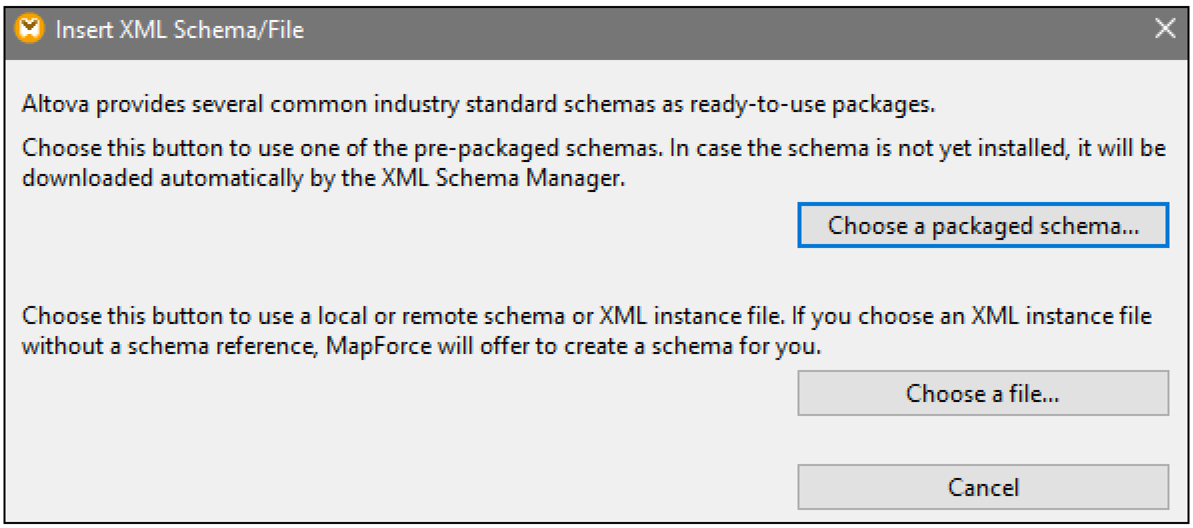
4.1 XML and XML Schema

Altova website: [XML Mapping]

This section provides information about aspects specific to XML components. For information about basic data transformation scenarios, see Tutorials[31].

Insert XML schema/file

To insert an XML schema/file, select the menu command **Insert | XML Schema/File** or the toolbar button. The dialog box (*see screenshot below*) will prompt you to choose between a packaged industry-standard schema and a local or remote schema/instance file. If you choose a packaged schema, you will be prompted to select an entry point. If the schema you wish to use is not yet installed, it will automatically be downloaded by the XML Schema Manager[122].



Generate an XML schema
When you add a local or remote XML file without a schema reference, MapForce will suggest generating an XML schema for you. 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.

DTD as a document structure
Starting with MapForce 2006 SP2, namespace-aware DTDs are supported for source and target components. To make mappings possible, the namespace-URIs are extracted from the DTD `xmlns` attribute declarations. However, some DTDs contain `xmlns*` attribute declarations without namespace URIs (e.g., DTDs used by StyleVision). To make such DTDs usable in MapForce, define the `xmlns` attribute with the namespace URI as follows:


```
<!ATTLIST fo:root
   xmlns:fo CDATA #FIXED 'http://www.w3.org/1999/XSL/Format'
   ...
>
```

In this section

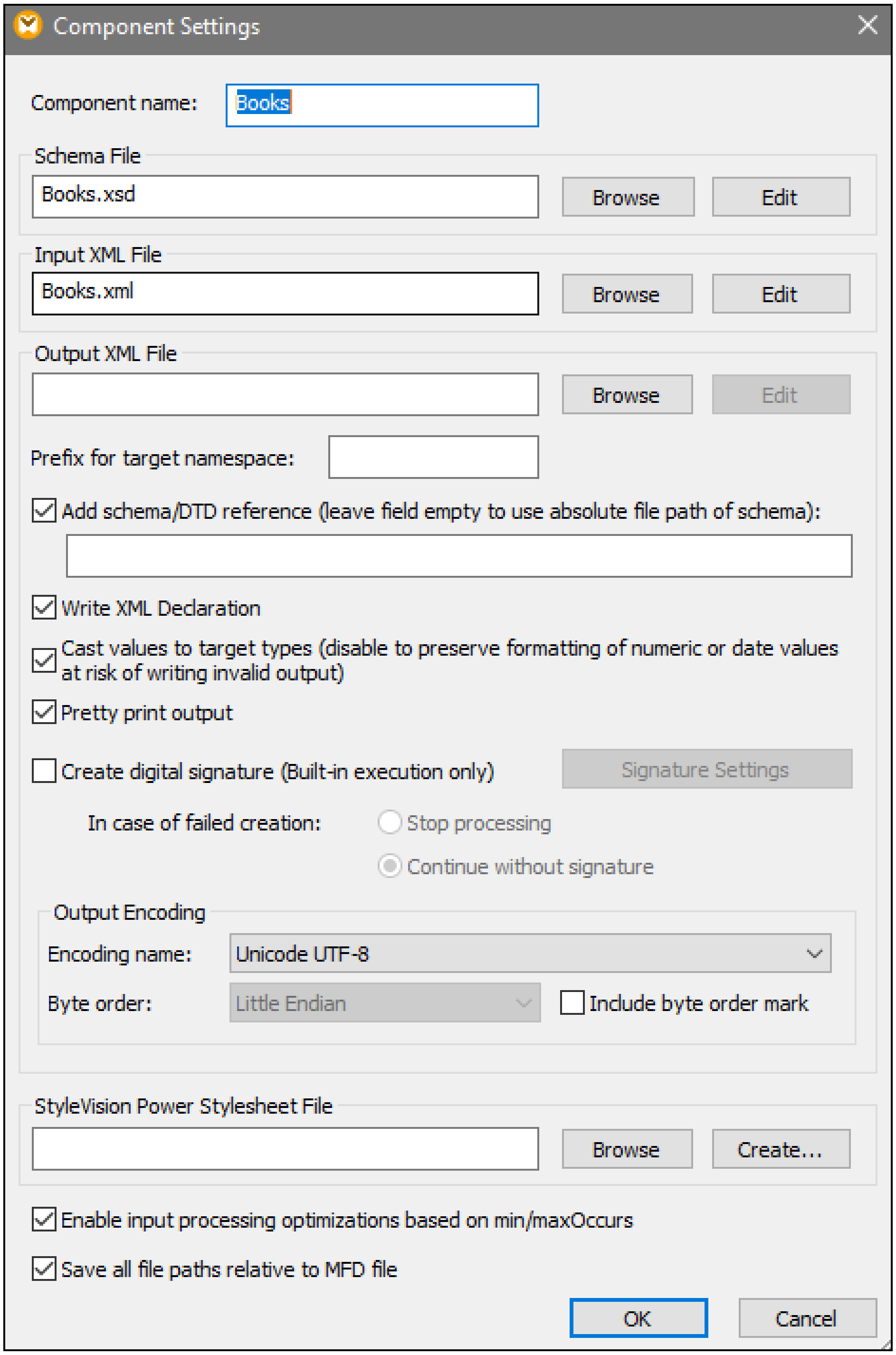
The section is organized into the following topics:

- XML Component Settings 107
- Derived Types 111
- NULL Values 113
- Comments and Processing Instructions 115
- CDATA Sections 116
- Wildcards: xs:any/xs:anyAttribute 117
- Custom Namespaces 120
- XML Schema Manager 122

4.1.1 XML Component Settings

After you add an XML component to the mapping area, you can configure its settings in the **Component Settings** dialog box (*see screenshot below*). You can open the **Component Settings** dialog box in one of the following ways:

- By double-clicking the component header.
- By right-clicking the component header and selecting **Properties**.
- By selecting the component in the mapping and clicking **Properties** in the **Component** menu.



The available settings are described in the subsections below.

General settings

⊟ Component name

The component name is automatically generated when you create a component. However, you can change the name at any time. The component name can contain spaces and full stop characters. It may not contain slashes, backslashes, colons, double quotes, leading and trailing spaces. If you want to change the name of the component, be aware of the following:

- If you intend to deploy the mapping to FlowForce Server, the component name must be unique.
- It is recommended to use only characters that can be entered at the command line. National characters may have a different encoding in Windows and at the command line.

Schema File

Specifies the name or path of the XML schema file used by MapForce to validate and map data. To change the schema file, click **Browse** and select a new file. To edit the file in Altova XMLSpy, click **Edit**.

Input XML File

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 an XML instance file to this component. 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. To change the schema file, click **Browse** and select a new file. To edit the file in Altova XMLSpy, click **Edit**.

Output XML File

Specifies the XML instance file to which MapForce will write data. This field is meaningful for a target component. To change the schema file, click **Browse** and select a new file. To edit the file in Altova XMLSpy, click **Edit**.

Prefix for target namespace

Allows you to enter a prefix for the target namespace. Before assigning the prefix, make sure the target namespace is defined in the target schema.

Add schema/DTD reference

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 file(s) in the `xsi:schemaLocation` attribute or into the `DOCTYPE` declaration if a DTD is used.

MapForce Professional and Enterprise editions: If you generate code in XQuery or C++, adding the DTD reference is not supported.

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 **`http://`** address as well as an absolute or relative path in this field.

Deactivating this option allows you to disconnect the XML instance from the referenced XML schema or DTD. This may be useful, for example, if you want to send the XML output to someone who does not have access to the underlying XML schema.

Write XML declaration

By default, the option is enabled, which means that the XML declaration is written to the output. The table below shows how this feature is supported in MapForce target languages and execution engines.

Target language/Execution engine	When output is a file	When output is a string
Built-in (*Professional and Enterprise editions*)	Yes	Yes
MapForce Server (*Professional and Enterprise editions*)	Yes	Yes
XSLT, XQuery	Yes	No
Code generator (C++, C#, Java) (*Professional and Enterprise editions*)	Yes	Yes

Cast values to target types

This option allows you to define (i) whether the target XML schema types should be used in the mapping or (ii) whether all data mapped to the target component should be treated as string values. By default, this setting is enabled. 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. You can use mapping functions to format the number as a string in the required format and then map this string to the target.

Note that disabling this option will also disable the detection of invalid values, e.g. writing letters into numeric fields.

Pretty print output

Reformats the output XML document to give it a structured look. Each child node is offset from its parent by a single tab character.

Create digital signature (*Enterprise Edition*)

Allows you to add a digital signature to the XML output instance file. Adding a digital signature is possible when you select Built-In as a transformation language.

Output Encoding

Allows you to specify the following settings of the output instance file:

- Encoding name
- Byte order
- Whether the byte order mark (BOM) character should be included

By default, any new components have the encoding defined in the *Default encoding for new components* option. You can access this option from **Tools | Options** (*General* section).

If the mapping generates XSLT 1.0/2.0, activating the *Byte Order Mark* check box does not have any effect**,** as these languages do not support Byte Order Marks.

StyleVision Power Stylesheet file

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. See also Using Relative Paths on a Component 73.

Other settings

⊟ Enable input processing optimizations based on min/maxOccurs

This option allows special handling for sequences that are known to contain exactly one item, such as required attributes or child elements with minOccurs and maxOccurs="1". 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).

If the input data is **not valid** 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 **invalid input**, disable this check box.

⊟ Save all file paths relative to MFD file

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 Relative and Absolute Paths 73.

4.1.2 Derived Types

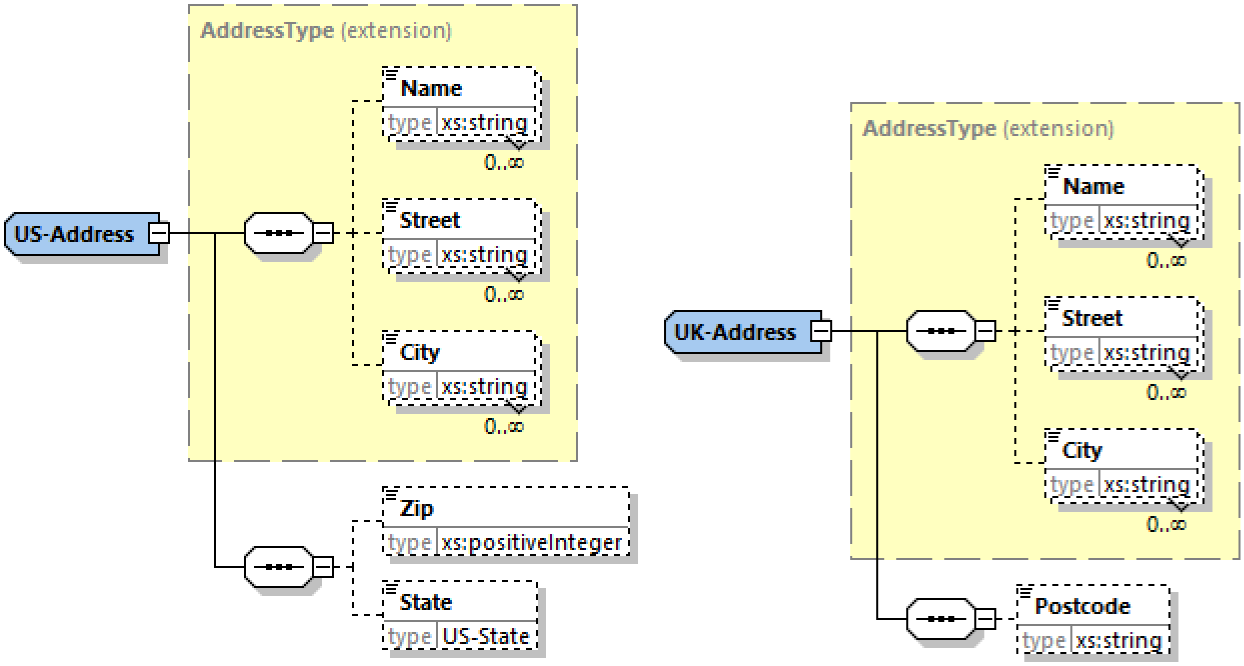
This topic explains how to use derived types in mappings. Derived types are defined in the W3C XML Schema Specification (Section 2.5.2). For a brief overview of primitive and derived types, see the Microsoft documentation. In order to use derived types in a mapping, you must specify the xsi:type attribute in your XML file (e.g., <Address xsi:type="UK-Address">).

Possible scenario

This subsection describes a possible scenario of using a derived type. For example, we have a company with two branches: one in the UK and the other in the US. Now we would like to have two lists (UKCustomers and USCustomers), each of which will include information about the respective branch address and all the customers associated with this branch.

Definition of derived types

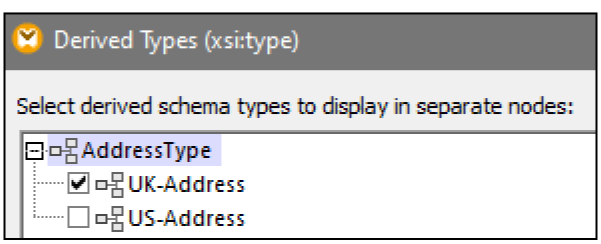
The screenshots below illustrate the definition of derived types called US-Address and UK-Address (*XMLSpy Schema view*). The UK-Address and UK-Address elements have the same base type called AddressType that includes the Name, Street, and City elements. In the US-Address element, the base type has been extended to include Zip and State, whereas the UK-Address element includes the base type and the Postcode element. For illustration purposes, we will map only the UK-Address element to the target file.



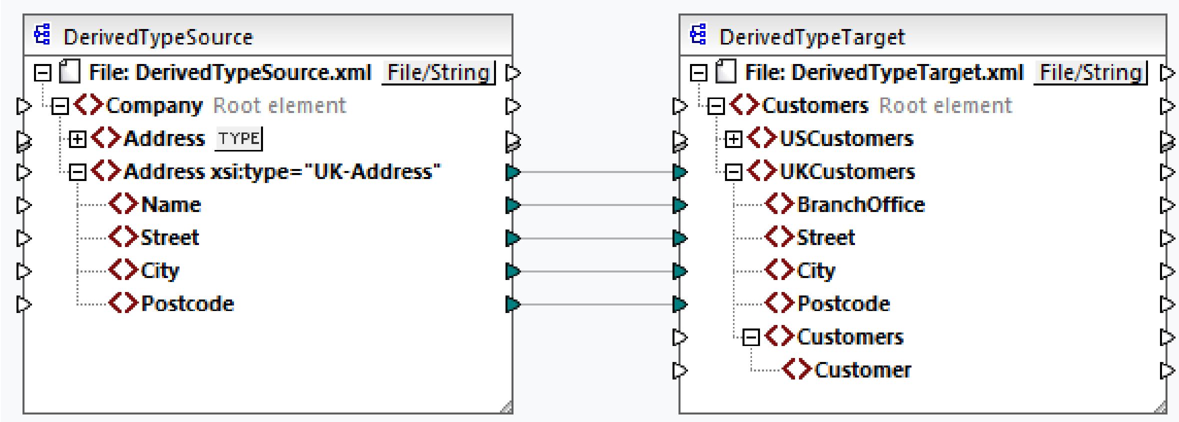
Derived type in a mapping

The instructions below show how to map data from the derived type. Our goal is to map information about the UK office to the `UKCustomers` element. The sample files are available in the **`Tutorial`** folder.

1. Go to the **Insert** menu, click **XML Schema/File**, and open **`DerivedTypeSource.xml`**. This XML file is based on **`DerivedTypeSource.xsd`**.
2. Insert the target file called **`DerivedTypeTarget.xsd`**. Note that the target schema does not have to include the `xsi:type` attribute.
3. Click the  button next to the `Address` element in the source component. This button indicates that derived types exist for this element in the schema.
4. The **Derived Types** dialog box (*see screenshot below*) allows you to select any derived types available for this specific element. In our sample mapping, we want only `UK-Address` to be mapped.



5. As soon as you select the check box next to the `UK-Address` derived type, a new element called `Address xsi:type="UK-Address"` appears in the component.
6. Now connect the nodes as shown in the mapping below.



Output

Clicking the **Output** pane will show the following result:

```
<UKCustomers>
   <BranchOffice>Sleuth Corp. UK</BranchOffice>
   <Street>222 Baker St</Street>
   <City>London</City>
   <Postcode>NW1 6XE</Postcode>
</UKCustomers>
```

The sample mapping is saved as **`Tutorial\DerivedType.mfd`**. You can also add another source XML file that includes information about the customers in the UK and map this data to the `Customers` node in the target component. This way, the `UKCustomers` element will include information about the UK address and all the customers associated with this branch.

4.1.3 NULL Values

This section describes how MapForce handles NULL values in source and target components. To be able to use the `xsi:nil="true"` attribute in your XML file, you must specify the `nillable="true"` attribute for the relevant element(s) in your schema file. To find out more about the `nillable` and `xsi:nil` attributes, see the W3C Specification. Note that the `xsi:nil` attribute is not visible in a component's tree in the **Mapping** pane.

The subsections below describe some of the possible scenarios of mapping NULL values.

NULL values in XML components

This subsection discusses some of the possible scenarios of mapping elements with an `xsi:nil="true"` attribute.

Only the source element has xsi:nil="true"/Both source and target elements have xsi:nil="true"

This scenario has the following conditions:

- The connection is target-driven [81].
- The source element has an `xsi:nil="true"` attribute. The corresponding target element does not have this attribute.
- Alternatively, both source and target elements can have `xsi:nil="true"` attributes.

- The `nillable="true"` attributes must be set in the source and target schemas.
- The source and target element are of simple type.

In this case, the target element will have the `xsi:nil="true"` attribute in the output file, as shown in the sample output file below (*highlighted in yellow*).

```
<book id="7">
	<author>Edgar Allan Poe</author>
	<title>The Murders in the Rue Morgue</title>
	<category xsi:nil="true"/>
	<year>1841</year>
	<OrderID id="213"/>
</book>
```

Note: If the `nillable="true"` attribute is *not* set in the target schema, the corresponding target element will be empty in the output.

Only the target element has xsi:nil="true"
This scenario has the following conditions:

- The connection is target-driven 81.
- The source element does not have an `xsi:nil="true"` attribute.
- The corresponding target element has an `xsi:nil="true"` attribute.
- The source and target elements can be of simple or complex type.

In this case, the source element will overwrite the target element containing the `xsi:nil="true"` attribute. The example below shows a sample output file. The `<genre>` element includes the `xsi:nil="true"` attribute in the target element. However, this element has been overwritten at mapping runtime. Therefore, the `<genre>` element (*highlighted in yellow*) has `Fiction` in the output.

```
<publication>
	<id>1</id>
	<author>Mark Twain</author>
	<title>The Adventures of Tom Sawyer</title>
	<genre>Fiction</genre>
	<year>1876</year>
	<OrderID id="124"/>
</publication>
```

Complex-type source element/both complex-type elements have xsi:nil="true"
This scenario has the following conditions:

- The connection is target-driven 81.
- The source element is of complex type. In our example, the source element has an `id="213"` attribute and an `xsi:nil="true"` attribute. The corresponding target element is also of complex type and has an `id="124"` attribute, but does not have an `xsi:nil="true"` attribute.
- Alternatively, the source and target elements, both of which are of complex type, can have `xsi:nil="true"` attributes.

In this case, the source element will overwrite the target element (*highlighted in yellow below*). However, the `xsi:nil="true"` attribute will not be written to the output file automatically. To see the `xsi:nil="true"` attribute in the target element in the output file, use a copy-all 86 connection.

```
<book id="7">
```

```
        <author>Edgar Allan Poe</author>
        <title>The Murders in the Rue Morgue</title>
        <year>1841</year>
        <OrderID id="213"/>
</book>
```

Useful functions

The following functions could help you check, replace, and assign NULL values:

- **is-xsi-nil** [263]: Helps to check explicitly whether a source element has a `xsi:nil` attribute set to `true`.
- **substitute-missing** [295]: Substitutes a NULL value in the source element with something specific.
- **set-xsi-nil** [265]: Assigns `xsi:nil="true"` attribute to a target element. This works for target elements of simple and complex types.
- **substitute-missing-with-xsi-nil** [267]: If there is content, it will be written to the target element; if there are any missing values, using this function will result in a target element with a `xsi:nil="true"` attribute in the output.
- Connecting the **exists** [271] function to a source element with a NULL value returns `true` even though the element has no content.

Note that functions which generate `xsi:nil` cannot be passed through functions or components which only operate on values (such as the **if-else** function).

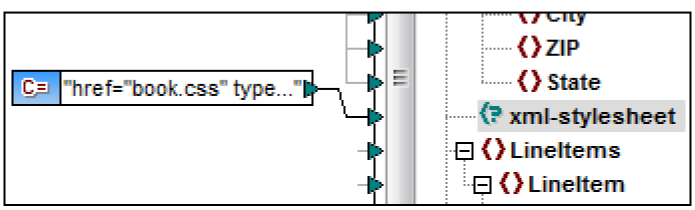
4.1.4 Comments and Processing Instructions

Comments and processing instructions are defined in the W3C Specification. This topic explains how to insert comments and processing instructions into target XML components. Note that comment and processing instruction nodes have only input connections. Comments and processing instructions cannot be defined for nodes that are part of a copy-all connection [86].

Insert a comment/processing instruction

To insert a processing instruction or a comment, take the steps below:

1. Right-click an element in the component and select **Add Comment/Processing Instruction Before/After**. When you insert a processing instruction, you will also need to enter its name. In the example below, a processing instruction called `xml-stylesheet` has been inserted after the `State` element.



3. To supply the value of a comment or a processing instruction, you can use a constant, for example (*see screenshot above*).

Note: Multiple processing instructions can be added before or after any element in the target component.

Note: Only one comment can be added before and after a single target node. To create multiple comments, use the duplicate input function 72.

Delete a comment/processing instruction
To delete a comment/processing instruction, right-click the respective node, select **Comment/Processing Instruction**, then select **Delete Comment/Processing Instruction** in the context menu.

4.1.5 CDATA Sections

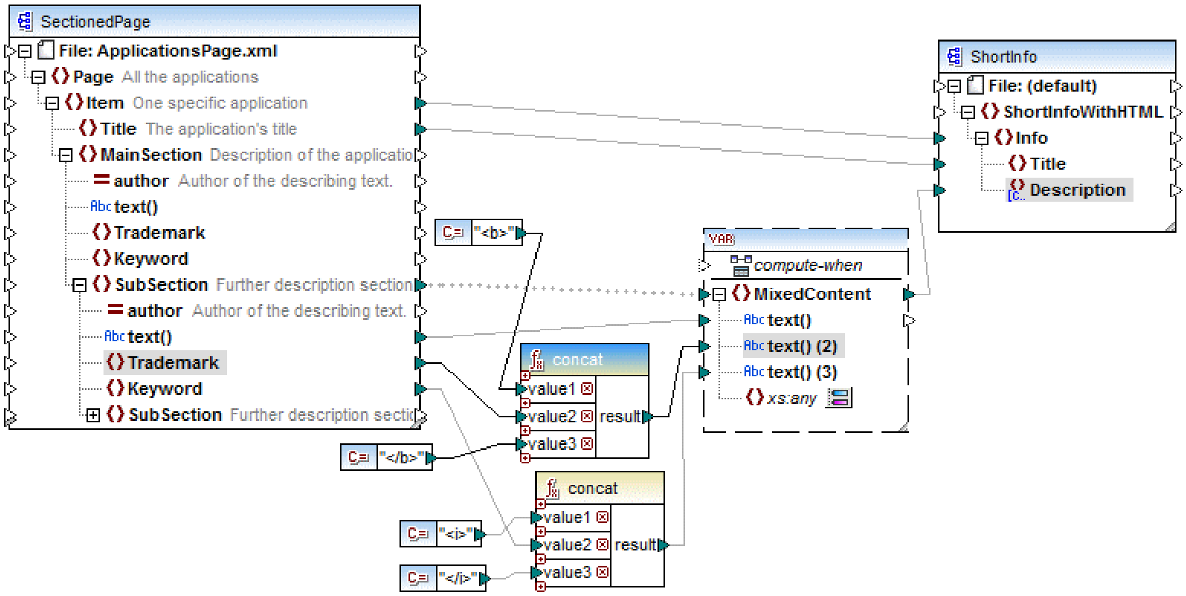
CDATA sections are used to represent parts of a document as character data which would normally be interpreted as markup. For more information about CDATA sections, see the W3C Specification. Target nodes receiving data as CDATA sections can be any of the following: XML data, XML data embedded in database fields, and XML child elements of typed dimensions in an XBRL target. CDATA sections can also be defined on duplicate nodes and `xsi:type` nodes.

To create a CDATA section, right-click the relevant target node and select **Write Content as CDATA Section**. A prompt will warn you that the input data should not contain the CDATA section close-delimiter **]]>**. The **[C..** icon appears below the element tag, which indicates that this node is now defined as a CDATA section.

Example

The example below shows a scenario in which a CDATA section might be useful. The sample mapping called **MapForceExamples\HTMLinCDATA.mfd** (*see screenshot below*) has the following aspects:

- The `SubSection` element has mixed content. For more information about mixed-content nodes, see Source-Driven Connections 82.
- With the help of the **concat** function, the content of the `Trademark` element will have the `<b></b>` tags.
- The content of the `Keyword` element will have the `<i></i>` tags.
- The data with the new tags is passed on to the duplicate `text()` nodes in the same order as in the source document.
- The output of the `MixedContent` node is then passed on to the `Description` node in the `ShortInfo` target component. The `Description` node has been defined as a CDATA section.



Output

Click the **Output** pane to see the CDATA section in the `Description` node (*screenshot below*).

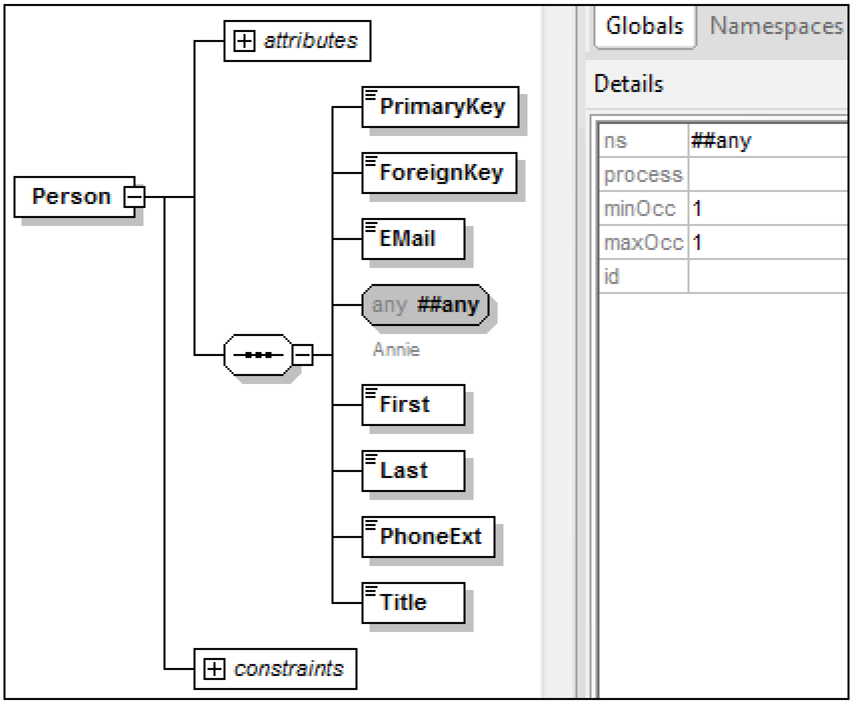
```
<Info>
    <Title>MapForce</Title>
    <Description><![CDATA[Altova  <b>MapForce</b>  2014 Enterprise Edition is the premier  <i>XML</i>
/ <i>database</i> / <i>flat file</i> / <i>EDI</i>  data mapping tool that auto-generates mapping code in
<i>XSLT</i>  1.0/2.0,  <i>XQuery</i> ,  <i>Java</i> ,  <i>C++</i>  and  <i>C#</i> . It is the definitive tool for
data integration and information leverage.]]></Description>
</Info>
```

4.1.6 Wildcards: xs:any/xs:anyAttribute

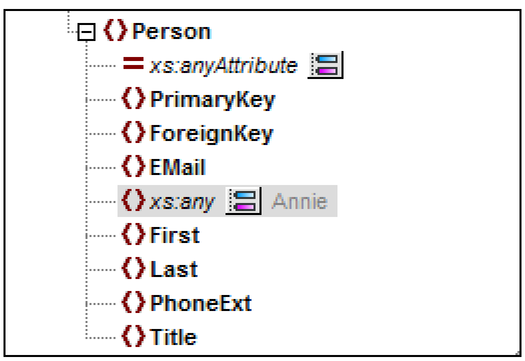
This topic explains how to deal with wildcards in mappings. The wildcards `xs:any` and `xs:anyAttribute` allow you to use any elements/attributes defined in your schema file. For more information about wildcards, see the W3C Specification.

Wildcards in schema definition

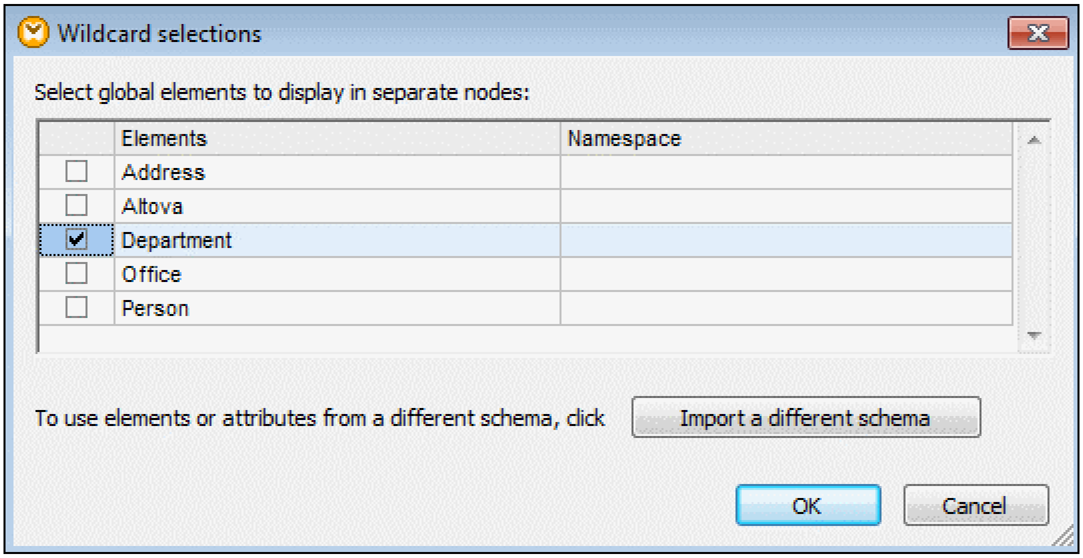
The screenshot below shows that an `xs:any` element has been defined as a child element of the `Person` element (*Schema view in Altova XMLSpy*).



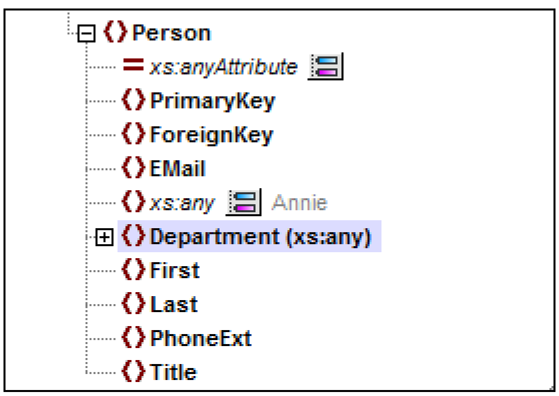
Wildcards in MapForce
When a wildcard is defined for an element and/or attribute, this wildcard node will have a (**Change Selection**) button next to it (*see screenshot below*).



Wildcard selection
Now our goal is to add another element as a separate node. Click the button to see the list of elements that you can add to the tree. Note that only elements and attributes that are globally declared in your schema can be seen in the **Wildcard selections** dialog box (*see screenshot below*).



For our example, we have selected `Department`. Note that wildcard elements and attributes are inserted after the node with the button. Now our component looks as follows:



You can now map to/from these nodes as usual. In a component, wildcard elements and attributes are marked with (`xs:any`) and (`xs:anyAttribute`), respectively (*see screenshot above*).

Remove wildcards
To remove a wildcard node, click the button and clear the corresponding check box in the **Wildcard selections** dialog box.

Elements/attributes from a different schema

The **Wildcard selections** dialog box (*see above*) allows you to use elements/attributes from a different schema. Clicking the **Import a different schema** button will give you the following options: (i) importing a schema file and (ii) generating a wrapper schema (*see description below*).

Import schema
The **Import schema** option imports the external schema into the current schema assigned to the component. Note that this option overrides the existing schema on the disk. If the current schema is a remote schema that

has been opened from a URL (see Adding Components from a URL 69) and not from the disk, the schema cannot be modified. In this case, use the **Generate wrapper schema** option.

Generate wrapper schema
The **Generate wrapper schema** option creates a new schema file called *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 which will include both the existing schema and the imported schema. When you select this option, you are prompted to choose where the wrapper schema should be saved. By default, the wrapper schema has the following format: **somefile-wrapper.xsd**.

After you save the wrapper schema, it is by default automatically assigned to the component. MapForce will also ask you whether you want to adjust the schema location so that you can reference the previous main schema. Click **Yes** to revert to the previous schema; otherwise, click **No** to have the newly created wrapper schema assigned to the component.

Wildcards vs. dynamic node names

There are situations in which elements and/or attributes in an instance are too many to be declared in the schema. Consider the following sample file:

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
   <line1>1</line1>
   <line2>2</line2>
   <line3>3</line3>
   ................
   <line999></line999>
</message>
```

For such situations, it is recommended to use dynamic node names instead of wildcards. For more information, see Mapping Node Names 380.

4.1.7 Custom Namespaces

When a mapping produces XML output, MapForce automatically derives the namespace (or set of namespaces) of each element and attribute from the target schema. This is the default behavior that is suitable for most scenarios of XML output generation. However, there are cases in which you may want to manually declare the namespace of an element directly from the mapping.

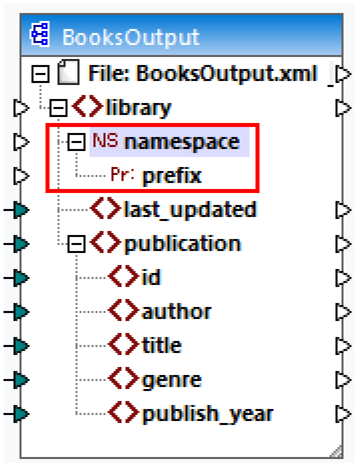
Declaring custom namespaces is meaningful only for target XML components and applies to elements only. The **Add Namespace** command is not available for attributes, wildcard nodes, and for nodes which receive data from a copy-all connection 86.

To understand how custom namespaces work, follow the instructions in the subsection below.

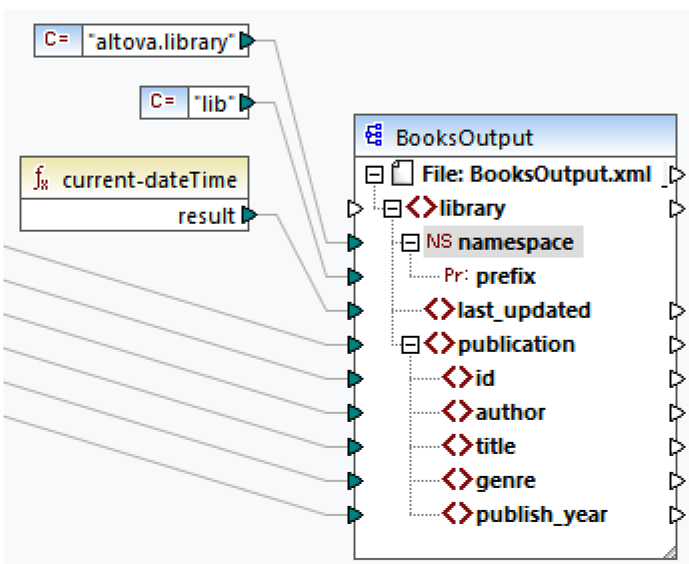
Declare namespace manually

For this example, you will need the following mapping: **BasicTutorials\Tut1-SchemaToSchema.mfd**.

Add a namespace
Open the mapping, right-click the `library` node in the `BooksOutput` component, and select **Add Namespace** from the context menu. Now two new nodes are available under the `library` element: `namespace` and `prefix` (*see screenshot below*).



Supply namespace values
The next step is to supply values to the `namespace` and `prefix` nodes. In order to do it, we will use two constants with the following string values: `altova.library` and `lib` (*see screenshot below*).



Note: Both the `namespace` and `prefix` input connectors must be mapped even if you provide empty values to them.

Output
In the generated output, an `xmlns:<prefix>="<namespace>"` attribute is added to the element, where `<prefix>` and `<namespace>` are values that are supplied by the mapping. The output will now look as follows (*note the highlighted part*):

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:lib="altova.library" xsi:noNamespaceSchemaLocation="Library.xsd">
...
```

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 new prefix and namespace values.

Declare a default namespace
If you want to declare a default namespace, map an empty string value to `prefix`. The output would then looks as follows (*note the highlighted part*):

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns="altova.library" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="Library.xsd">
...
```

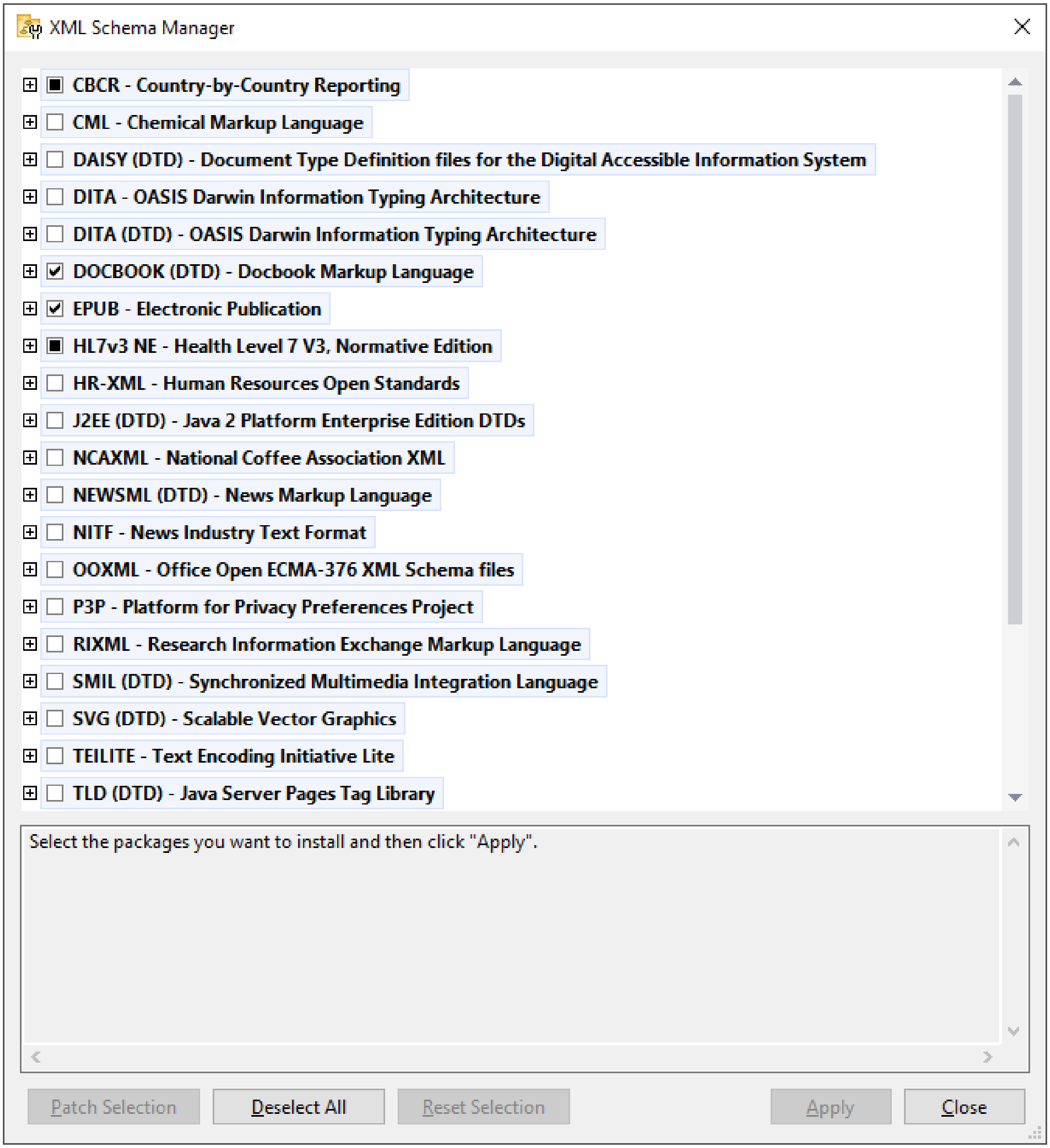
If you need to create prefixes for attribute names, for example `<number prod:id="prod557">557</number>`, you can achieve this by using dynamic access to a node's attributes (see Mapping Node Names 380) or by editing the schema so that it has a `prod:id` attribute for `<number>`.

Remove a namespace
To remove a previously added namespace declaration, right-click the `ns:namespace` node and select **Remove Namespace** from the context menu.

4.1.8 Schema Manager

XML Schema Manager is an Altova tool that provides a centralized way to install and manage XML schemas (DTDs for XML and XML Schemas) for use across all Altova's XBRL-enabled applications, including MapForce.

- On Windows, Schema Manager has a graphical user interface (*screenshot below*) and is also available at the command line. (Altova's desktop applications are available on Windows only; *see list below*.)
- On Linux and macOS, Schema Manager is available at the command line only. (Altova's server applications are available on Windows, Linux, and macOS; *see list below*.)



Altova applications that operate with Schema Manager

Desktop applications (Windows only)	Server applications (Windows, Linux, macOS)
XMLSpy (all editions)	RaptorXML Server, RaptorXML+XBRL Server

MapForce (all editions)	StyleVision Server
StyleVision (all editions)	
Authentic Desktop Enterprise Edition	

Installation and de-installation of Schema Manager

Schema Manager is installed automatically when you first install a new version of Altova Mission Kit Enterprise Edition or of any of Altova's XML-schema-aware applications (*see table above*).

Likewise, it is removed automatically when you uninstall the last Altova XML-schema-aware application from your computer.

Schema Manager features

Schema Manager provides the following features:

- Shows XML schemas installed on your computer and checks whether new versions are available for download.
- Downloads newer versions of XML schemas independently of the Altova product release cycle. (Altova stores schemas online, and you can download them via Schema Manager.)
- Install or uninstall any of the multiple versions of a given schema (or all versions if necessary).
- An XML schema may have dependencies on other schemas. When you install or uninstall a particular schema, Schema Manager informs you about dependent schemas and will automatically install or remove them as well.
- Schema Manager uses the XML catalog mechanism to map schema references to local files. In the case of large XML schemas, processing will therefore be faster than if the schemas were at a remote location.
- All major schemas are available via Schema Manager and are regularly updated for the latest versions. This provides you with a convenient single resource for managing all your schemas and making them readily available to all of Altova's XML-schema-aware applications.
- Changes made in Schema Manager take effect for all Altova products installed on that machine.

How it works

Altova stores all XML schemas used in Altova products online. This repository is updated when new versions of the schemas are released. Schema Manager displays information about the latest available schemas when invoked in both its GUI form as well as on the CLI. You can then install, upgrade or uninstall schemas via Schema Manager.

Schema Manager also installs schemas in one other way. At the Altova website (https://www.altova.com/schema-manager) you can select a schema and its dependent schemas that you want to install. The website will prepare a file of type **`.altova_xmlschemas`** for download that contains information about your schema selection. When you double-click this file or pass it to Schema Manager via the CLI as an argument of the `install` 133 command, Schema Manager will install the schemas you selected.

Local cache: tracking your schemas

All information about installed schemas is tracked in a centralized cache directory on your computer, located here:

Windows	`C:\ProgramData\Altova\pkgs\.cache`

Linux	`/var/opt/Altova/pkgs\.cache`
macOS	`/var/Altova/pkgs`

This cache directory is updated regularly with the latest status of schemas at Altova's online storage. These updates are carried out at the following times:

- Every time you start Schema Manager.
- When you start MapForce for the first time on a given calendar day.
- If MapForce is open for more than 24 hours, the cache is updated every 24 hours.
- You can also update the cache by running the `update` 136 command at the command line interface.

The cache therefore enables Schema Manager to continuously track your installed schemas against the schemas available online at the Altova website.

Do not modify the cache manually!

The local cache directory is maintained automatically based on the schemas you install and uninstall. It should not be altered or deleted manually. If you ever need to reset Schema Manager to its original "pristine" state, then, on the command line interface (CLI): (i) run the `reset` 134 command, and (ii) run the `initialize` 132 command. (Alternatively, run the **`reset`** command with the **`--i`** option.)

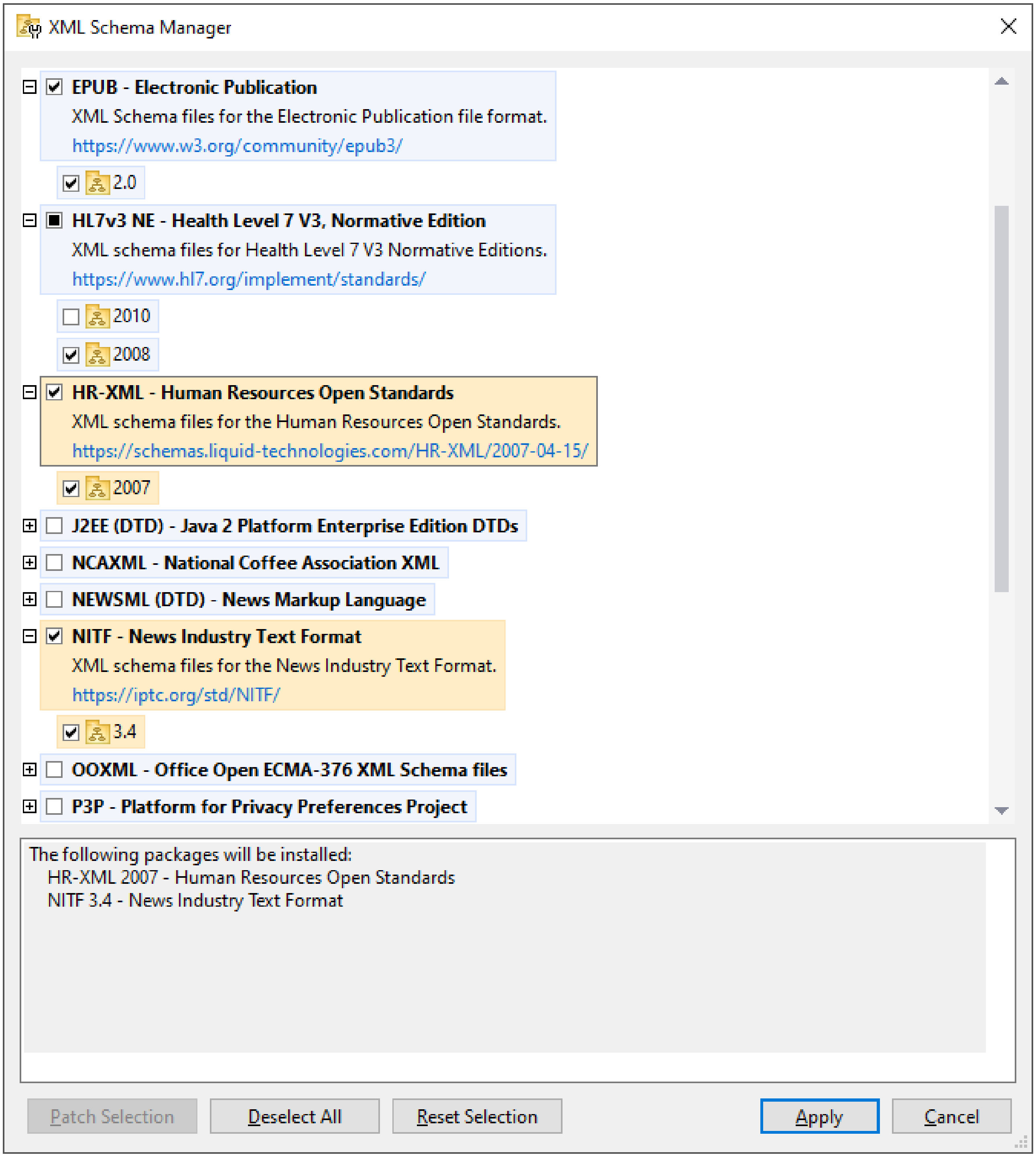
4.1.8.1 Run Schema Manager

Graphical User Interface

You can access the GUI of Schema Manager in any of the following ways:

- *During the installation of MapForce:* Towards the end of the installation procedure, select the check box *Invoke Altova XML-Schema Manager* to access the Schema Manager GUI straight away. This will enable you to install schemas during the installation process of your Altova application.
- *After the installation of MapForce:* After your application has been installed, you can access the Schema Manager GUI at any time, via the menu command **Tools | XML Schema Manager**.
- Via the **`.altova_xmlschemas`** file downloaded from the Altova website: Double-click the downloaded file to run the Schema Manager GUI, which will be set up to install the schemas you selected (at the website) for installation.

After the Schema Manager GUI (*screenshot below*) has been opened, already installed schemas will be shown selected. If you want to install an additional schema, select it. If you want to uninstall an already installed schema, deselect it. After you have made your selections and/or deselections, you are ready to apply your changes. The schemas that will be installed or uninstalled will be highlighted and a message about the upcoming changes will be posted to the Messages pane at the bottom of the Schema Manager window (*see screenshot*).



Command line interface

You can run Schema Manager from a command line interface by sending commands to its executable file, `xmlschemamanager.exe`.

The **`xmlschemamanager.exe`** file is located in the following folder:

- *On Windows:* `C:\ProgramData\Altova\SharedBetweenVersions`
- *On Linux or macOS (server applications only):* `%INSTALLDIR%/bin`, where `%INSTALLDIR%` is the program's installation directory.

You can then use any of the commands listed in the CLI command reference section 131.

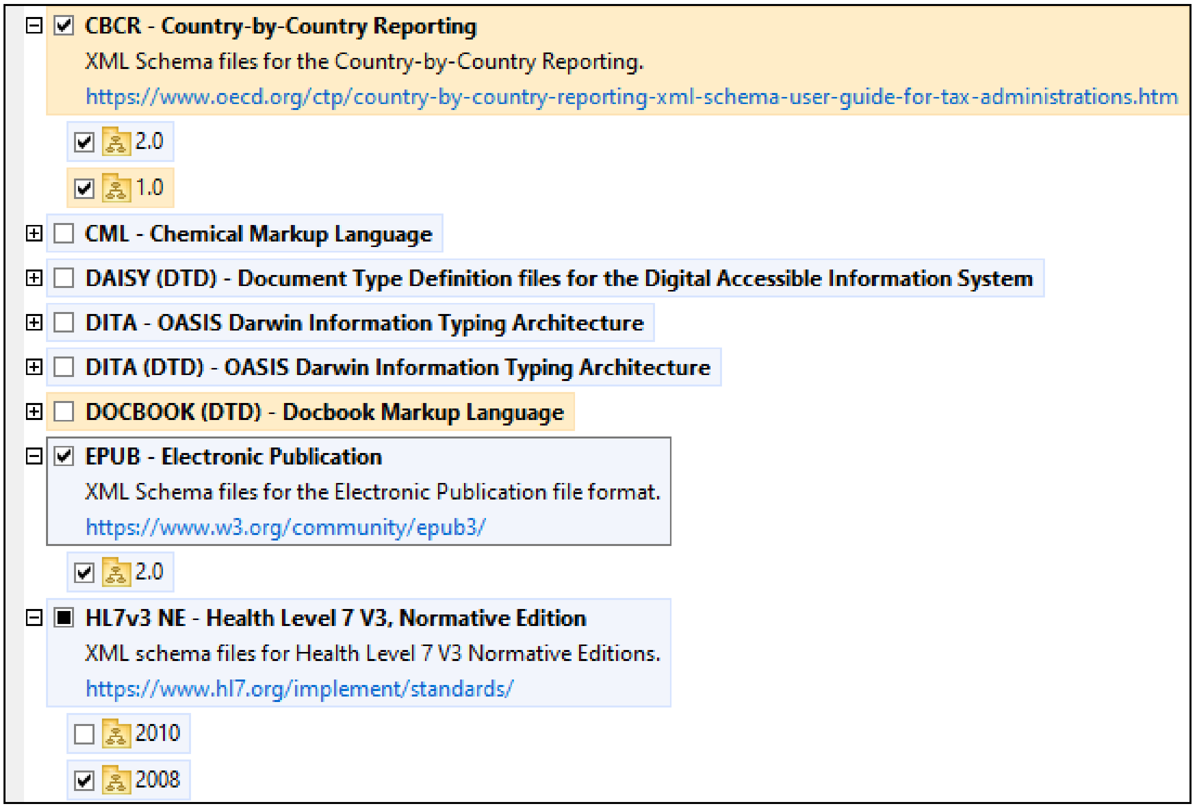
To display help for the commands, run the following:

- *On Windows:* `xmlschemamanager.exe --help`
- *On Linux or macOS (server applications only):* `sudo ./xmlschemamanager --help`

4.1.8.2 Status Categories

Schema Manager categorizes the schemas under its management as follows:

- *Installed schemas.* These are shown in the GUI with their check boxes selected (*in the screenshot below the checked and blue versions of the EPUB and HL7v3 NE schemas are installed schemas*). If all the versions of a schema are selected, then the selection mark is a tick. If at least one version is unselected, then the selection mark is a solid colored square. You can deselect an installed schema to **uninstall** it; (*in the screenshot below, the DocBook DTD is installed and has been deselected, thereby preparing it for de-installation*).
- *Uninstalled available schemas.* These are shown in the GUI with their check boxes unselected. You can select the schemas you want to **install**.



- *Upgradeable schemas* are those which have been revised by their issuers since they were installed. They are indicated in the GUI by a icon. You can **patch** an installed schema with an available revision.

Points to note

- In the screenshot above, both CBCR schemas are checked. The one with the blue background is already installed. The one with the yellow background is uninstalled and has been selected for installation. Note that the HL7v3 NE 2010 schema is not installed and has not been selected for installation.
- A yellow background means that the schema will be modified in some way when the **Apply** button is clicked. If a schema is unchecked and has a yellow background, it means that it will be uninstalled when the **Apply** button is clicked. In the screenshot above the DocBook DTD has such a status.
- When running Schema Manager from the command line, the `list` 133 command is used with different options to list different categories of schemas:

`xmlschemamanager.exe list`	Lists all installed and available schemas; upgradeables are also indicated
`xmlschemamanager.exe list -i`	Lists installed schemas only; upgradeables are also indicated
`xmlschemamanager.exe list -u`	Lists upgradeable schemas

Note: On Linux and macOS, **`use sudo ./xmlschemamanager list`**

4.1.8.3 Patch or Install a Schema

Patch an installed schema

Occasionally, XML schemas may receive patches (upgrades or revisions) from their issuers. When Schema Manager detects that patches are available, these are indicated in the schema listings of Schema Manager and you can install the patches quickly.

In the GUI

Patches are indicated by the icon. (*Also see the previous topic about status categories*[127].) If patches are available, the **Patch Selection** button will be enabled. Click it to select and prepare all patches for installation. In the GUI, the icon of each schema that will be patched changes from to , and the Messages pane at the bottom of the dialog lists the patches that will be applied. When you are ready to install the selected patches, click **Apply**. All patches will be applied together. Note that if you deselect a schema marked for patching, you will actually be uninstalling that schema.

On the CLI

To apply a patch at the command line interface:

1. Run the `list -u`[133] command. This lists any schemas for which upgrades are available.
2. Run the `upgrade`[136] command to install all the patches.

Install an available schema

You can install schemas using either the Schema Manager GUI or by sending Schema Manager the install instructions via the command line.

Note: If the current schema references other schemas, the referenced schemas are also installed.

In the GUI

To install schemas using the Schema Manager GUI, select the schemas you want to install and click **Apply**.

You can also select the schemas you want to install at the Altova website and generate a downloadable **`.altova_xmlschemas`** file. When you double-click this file, it will open Schema Manager with the schemas you wanted pre-selected. All you will now have to do is click **Apply**.

On the CLI

To install schemas via the command line, run the `install`[133] command:

```
xmlschemamanager.exe install [options] Schema+
```

where **`Schema`** is the schema (or schemas) you want to install or a **`.altova_xmlschemas`** file. A schema is referenced by an identifier of format **`<name>-<version>`**. (The identifiers of schemas are displayed when you run the `list`[133] command.) You can enter as many schemas as you like. For details, see the description of the `install`[133] command.

Note: On Linux or macOS, use the **`sudo ./xmlschemamanager`** command.

Installing a required schema
When you run an XML-schema-related command in MapForce and MapForce discovers that a schema it needs for executing the command is not present or is incomplete, Schema Manager will display information about the missing schema/s. You can then directly install any missing schema via Schema Manager.

In the Schema Manager GUI, you can view all previously installed schemas at any time by running Schema Manager from **Tools | Schema Manager**.

4.1.8.4 Uninstall a Schema, Reset

Uninstall a schema

You can uninstall schemas using either the Schema Manager GUI or by sending Schema Manager the uninstall instructions via the command line.

Note: If the schema you want to uninstall references other schemas, then the referenced schemas are also uninstalled.

In the GUI
To uninstall schemas in the Schema Manager GUI, clear their check boxes and click **Apply**. The selected schemas and their referenced schemas will be uninstalled.

To uninstall all schemas, click **Deselect All** and click **Apply**.

On the CLI
To uninstall schemas via the command line, run the `uninstall` 135 command:

```
xmlschemamanager.exe uninstall [options] Schema+
```

where each **`Schema`** argument is a schema you want to uninstall or a **`.altova_xmlschemas`** file. A schema is specified by an identifier that has a format of **`<name>-<version>`**. (The identifiers of schemas are displayed when you run the `list` 133 command.) You can enter as many schemas as you like. For details, see the description of the `uninstall` 135 command.

Note: On Linux or macOS, use the **`sudo ./xmlschemamanager`** command.

Reset Schema Manager

You can reset Schema Manager. This removes all installed schemas and the cache directory.

- In the GUI, click **Reset Selection**.
- On the CLI, run the `reset` 134 command.

After running this command, make sure to run the `initialize` 132 command in order to recreate the cache directory. Alternatively, run the `reset` 134 command with the `-i` option.

Note that `reset -i` 134 restores the original installation of the product, so it is recommended to run the `update` 136 command after performing a reset. Alternatively, run the `reset` 134 command with the `-i` and `-u` options.

4.1.8.5 Command Line Interface (CLI)

To call Schema Manager at the command line, you need to know the path of the executable. By default, the Schema Manager executable is installed here:

```
C:\ProgramData\Altova\SharedBetweenVersions\XMLSchemaManager.exe
```

Note: On Linux and macOS systems, once you have changed the directory to that containing the executable, you can call the executable with **`sudo ./xmlschemamanager`**. The prefix **`./`** indicates that the executable is in the current directory. The prefix **`sudo`** indicates that the command must be run with root privileges.

Command line syntax

The general syntax for using the command line is as follows:

```
<exec> -h | --help | --version | <command> [options] [arguments]
```

In the listing above, the vertical bar **`|`** separates a set of mutually exclusive items. The square brackets **`[]`** indicate optional items. Essentially, you can type the executable path followed by either `--h`, `--help`, or `--version` options, or by a command. Each command may have options and arguments. The list of commands is described in the following sections.

4.1.8.5.1 help

This command provides contextual help about commands pertaining to Schema Manager executable.

Syntax

```
<exec> help [command]
```

Where `[command]` is an optional argument which specifies any valid command name.

Note the following:

- You can invoke help for a command by typing the command followed by **`-h`** or **`--help`**, for example: **`<exec> list -h`**
- If you type **`-h`** or **`--help`** directly after the executable and before a command, you will get general help (not help for the command), for example: **`<exec> -h list`**

Example

The following command displays help about the `list` command:

```
xmlschemamanager help list
```

4.1.8.5.2 info

This command displays detailed information for each of the schemas supplied as a `Schema` argument. This information for each submitted schema includes the title, version, description, publisher, and any referenced schemas, as well as whether the schema has been installed or not.

Syntax

```
<exec> info [options] Schema+
```

- The **`Schema`** argument is the name of a schema or a part of a schema's name. (To display a schema's package ID and detailed information about its installation status, you should use the `list` 133 command.)
- Use **`<exec> info -h`** to display help for the command.

Example

The following command displays information about the latest **`DocBook-DTD`** and **`NITF`** schemas:

```
xmlschemamanager info doc nitf
```

4.1.8.5.3 initialize

This command initializes the Schema Manager environment. It creates a cache directory where information about all schemas is stored. Initialization is performed automatically the first time a schema-cognizant Altova application is installed. You would not need to run this command under normal circumstances, but you would typically need to run it after executing the **`reset`** command.

Syntax

```
<exec> initialize | init [options]
```

Options

The **`initialize`** command takes the following options:

`--silent, --s`	Display only error messages. The default is **`false`**.
`--verbose, --v`	Display detailed information during execution. The default is **`false`**.
`--help, --h`	Display help for the command.

Example

The following command initializes Schema Manager:

```
xmlschemamanager initialize
```

4.1.8.5.4 install

This command installs one or more schemas.

Syntax

```
<exec> install [options] Schema+
```

To install multiple schemas, add the **Schema** argument multiple times.

The **Schema** argument is one of the following:

- A schema identifier (having a format of **<name>-<version>**, for example: **cbcr-2.0**). To find out the schema identifiers of the schemas you want, run the list133 command. You can also use an abbreviated identifier if it is unique, for example **docbook**. If you use an abbreviated identifier, then the latest version of that schema will be installed.
- The path to a **.altova_xmlschemas** file downloaded from the Altova website. For information about these files, see *Introduction to SchemaManager: How It Works*122.

Options

The **install** command takes the following options:

`--silent, --s`	Display only error messages. The default is **false**.
`--verbose, --v`	Display detailed information during execution. The default is **false**.
`--help, --h`	Display help for the command.

Example

The following command installs the CBCR 2.0 (Country-By-Country Reporting) schema and the latest DocBook DTD:

```
xmlschemamanager install cbcr-2.0 docbook
```

4.1.8.5.5 list

This command lists schemas under the management of Schema Manager. The list displays one of the following

- All available schemas
- Schemas containing in their name the string submitted as a **`Schema`** argument
- Only installed schemas
- Only schemas that can be upgraded

Syntax

```
<exec> list | ls [options] Schema?
```

If no **`Schema`** argument is submitted, then all available schemas are listed. Otherwise, schemas are listed as specified by the submitted options (*see example below*). Note that you can submit the **`Schema`** argument multiple times.

Options
The **`list`** command takes the following options:

`--installed, --i`	List only installed schemas. The default is **`false`**.
`--upgradeable, --u`	List only schemas where upgrades (patches) are available. The default is **`false`**.
`--help, --h`	Display help for the command.

Examples

- To list all available schemas, run: **`xmlschemamanager list`**
- To list installed schemas only, run: **`xmlschemamanager list -i`**
- To list schemas that contain either "doc" or "nitf" in their name, run: **`xmlschemamanager list doc nitf`**

4.1.8.5.6 reset

This command removes all installed schemas and the cache directory. You will be completely resetting your schema environment. After running this command, be sure to run the `initialize` 132 command to recreate the cache directory. Alternatively, run the **`reset`** command with the **`-i`** option. Since **`reset -i`** restores the original installation of the product, we recommend that you run the `update` 136 command after performing a reset and initialization. Alternatively, run the **`reset`** command with both the **`-i`** and **`-u`** options.

Syntax

```
<exec> reset [options]
```

Options
The **`reset`** command takes the following options:

`--init, --i`	Initialize Schema Manager after reset. The default is **`false`**.
`--update, --u`	Updates the list of available schemas in the cache. The default is **`false`**.

`--silent, --s`	Display only error messages. The default is **false**.
`--verbose, --v`	Display detailed information during execution. The default is **false**.
`--help, --h`	Display help for the command.

Examples

- To reset Schema Manager, run: **xmlschemamanager reset**
- To reset Schema Manager and initialize it, run: **xmlschemamanager reset -i**
- To reset Schema Manager, initialize it,and update its schema list, run: **xmlschemamanager reset -i -u**

4.1.8.5.7 uninstall

This command uninstalls one or more schemas. By default, any schemas referenced by the current one are uninstalled as well. To uninstall just the current schema and keep the referenced schemas, set the option `--k`.

Syntax

```
<exec> uninstall [options] Schema+
```

To uninstall multiple schemas, add the **Schema** argument multiple times.

The **Schema** argument is one of the following:

- A schema identifier (having a format of **<name>-<version>**, for example: **cbcr-2.0**). To find out the schema identifiers of the schemas that are installed, run the `list -i` 133 command. You can also use an abbreviated schema name if it is unique, for example **docbook**. If you use an abbreviated name, then all schemas that contain the abbreviation in its name will be uninstalled.
- The path to a **.altova_xmlschemas** file downloaded from the Altova website. For information about these files, see *Introduction to SchemaManager: How It Works* 122.

Options
The **uninstall** command takes the following options:

`--keep-references, --k`	Set this option to keep referenced schemas. The default is **false**.
`--silent, --s`	Display only error messages. The default is **false**.
`--verbose, --v`	Display detailed information during execution. The default is **false**.
`--help, --h`	Display help for the command.

Example

The following command uninstalls the CBCR 2.0 and EPUB 2.0 schemas and their dependencies:

```
xmlschemamanager uninstall cbcr-2.0 epub-2.0
```

The following command uninstalls the **eba-2.10** schema but not the schemas it references:

```
xmlschemamanager uninstall --k cbcr-2.0
```

4.1.8.5.8 update

This command queries the list of schemas available from the online storage and updates the local cache directory. You should not need to run this command unless you have performed a `reset` 134 and `initialize` 132.

Syntax

```
<exec> update [options]
```

Options

The **update** command takes the following options:

`--silent, --s`	Display only error messages. The default is **false**.
`--verbose, --v`	Display detailed information during execution. The default is **false**.
`--help, --h`	Display help for the command.

Example

The following command updates the local cache with the list of latest schemas:

```
xmlschemamanager update
```

4.1.8.5.9 upgrade

This command upgrades all installed schemas that can be upgraded to the latest available *patched* version. You can identify upgradeable schemas by running the `list -u` 133 command.

Note: The **upgrade** command removes a deprecated schema if no newer version is available.

Syntax

```
<exec> upgrade [options]
```

Options

The **upgrade** command takes the following options:

`--silent, --s`	Display only error messages. The default is **false**.
`--verbose, --v`	Display detailed information during execution. The default is **false**.

`--help, --h`	Display help for the command.

5 Transformation Components

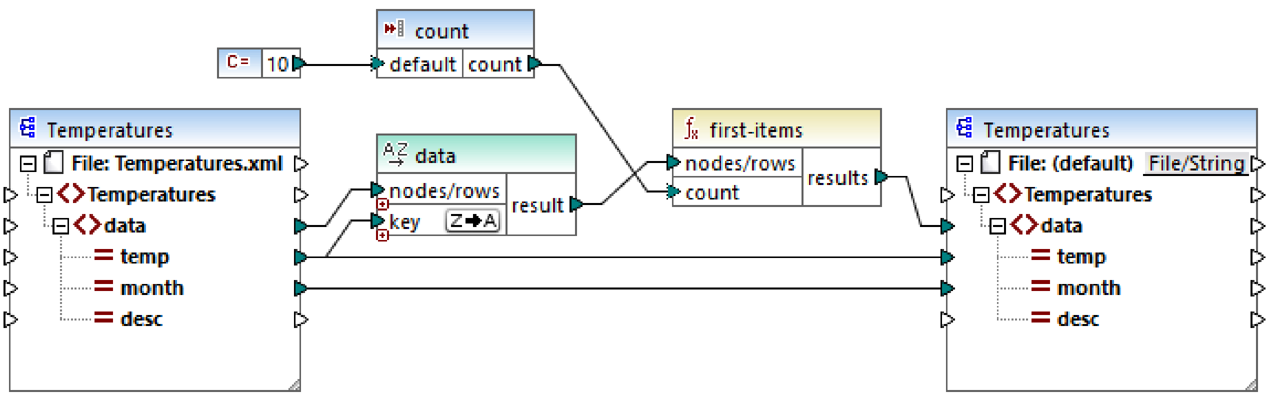
This section describes transformation components that can be used to transform data or to store data temporarily for further processing. The list of transformation components is given below:

- Simple input [139]
- Simple output [146]
- Variables [150]
- Sort Components [162]
- Filters and Conditions [168]
- Value-Maps [174]
- Group Functions [184]

Note that functions also belong to transformation components. However, functions [190] are organized as a standalone section.

5.1 Simple Input

If you need to create a mapping that takes parameters as input, you can do so by adding a special component type called "simple input component". Simple input components always have a simple data type (for example, string, integer, and so on) instead of a structure of items and sequences. For example, in the mapping illustrated below, there is a simple input component **count**. Its role is to supply as parameter the maximum number of rows that should be retrieved from the source XML file (with value **10** as default). Importantly, the nodes supplied as input to the first-items [273] function are sorted with the help of a sort component, so the mapping outputs the highest *N* temperatures only, where *N* is the parameter's value.



FindHighestTemperatures.mfd

Another fairly common usage of simple input components is to supply a file name to the mapping. This is useful in mappings that read input files or write output files dynamically, see Processing Multiple Input or Output Files Dynamically [417]. 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 Adding Simple Input Components [140]. If necessary, you can also create default values for the mapping input parameters, see Creating a Default Input Value [142]. This enables you to safely run the mapping even if you do not explicitly supply a parameter value at mapping execution time. For an example, see Example: Using File Names as Mapping Parameters [143].

Input parameters added on the main mapping area should not be confused with input parameters in user-defined functions [198]. There are some similarities and differences between the two, as follows.

Input parameters on the mapping	Input parameters of user-defined functions
Added from **Function | Insert Input** menu.	Added from **Function | Insert Input** 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.

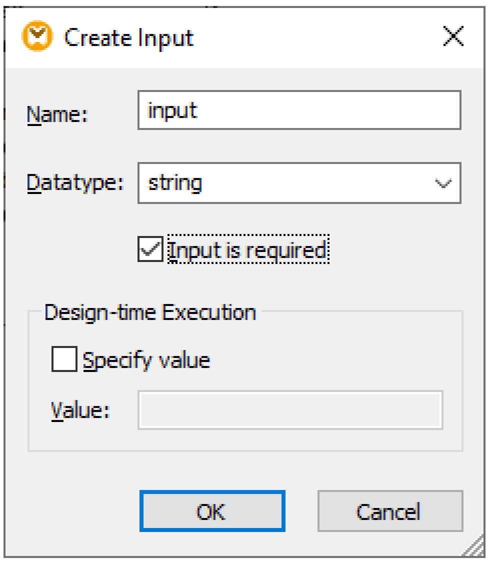
When you create a reversed mapping (using the menu command **Tools | Create Reversed Mapping**), a simple input component becomes a simple output component.

5.1.1 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. Do one of the following:

   - On the **Function** menu, click **Insert Input**.
   - On the **Insert** menu, click **Insert Input**.
   - Click the **Insert Input** toolbar button.



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 140.

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**.

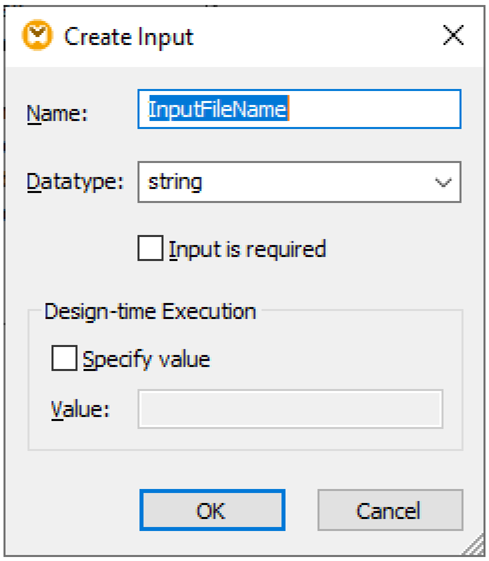
You can change later any of the settings defined here (see Simple Input Component Settings 140).

5.1.2 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.

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**.



Edit Input dialog box

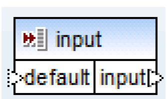
The available settings are as follows.

Name	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.
Datatype	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.
Input is required	When enabled, this setting makes the input parameter mandatory (that is, the mapping cannot be executed unless you supply a parameter value). Clear this check box if you want to specify a default value for the input parameter (see Creating a Default Input Value 142).
Specify value	This setting is applicable only if you execute the mapping during design time, by clicking the **Preview** tab. It allows you to enter directly in the component the value to use as mapping input.
Value	This setting is applicable only if you execute the mapping during design time, by clicking the **Preview** tab. To enter a value to be used by MapForce as mapping input, select the **Specify Value** check box, and then type the required value.

	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 design-time value has no effect in the generated XSLT, XQuery, or program code, in execution by MapForce Server, or deployment to FlowForce Server.

5.1.3 Creating a Default Input Value

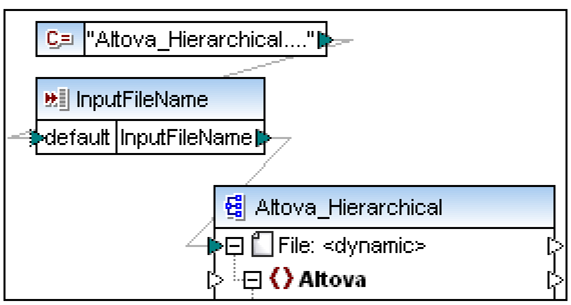
After you add an Input component to the mapping area, notice the **default** item to the left of the component.



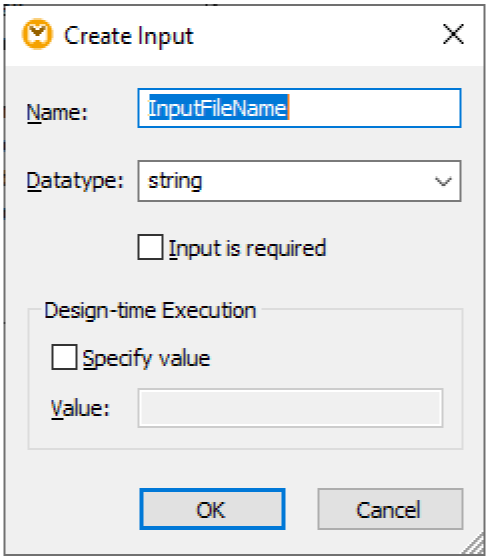
Simple input component

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 clear the **Input is required** check box. When you create a default input value, this setting is not meaningful and causes mapping validation warnings.



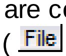
3. Click **OK**.

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 design-time value has no effect in the generated XSLT, XQuery, or program code, in execution by MapForce Server, or deployment to FlowForce Server.

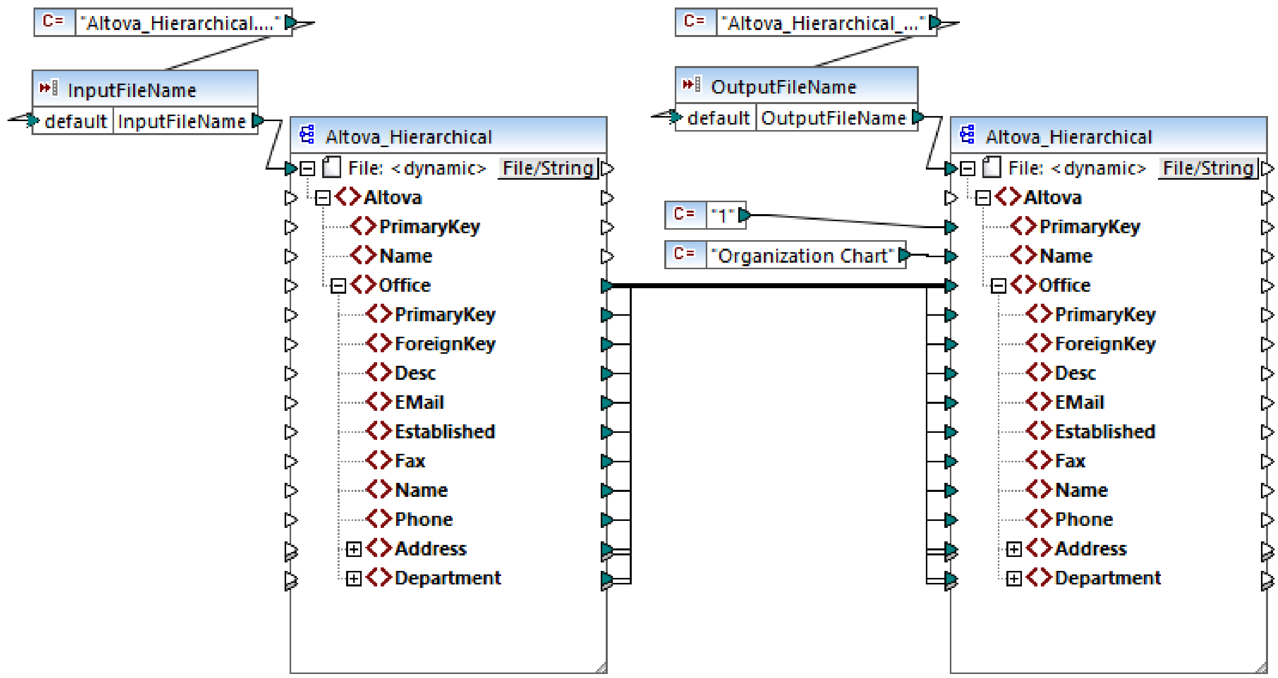
5.1.4 Example: Using File Names as Mapping Parameters

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\MapForce2023\MapForceExamples\FileNamesAsParameters.mfd**.

This mapping reads data from a source XML file and writes it to a target XML file. The data is written to the target file almost unchanged; only the attributes **PrimaryKey** and **Name** are populated with some constant values from the mapping. The main goal of the mapping is to enable the caller to specify the name of the input file and the name of the output file, as mapping parameters, at mapping runtime.

To achieve this, the mapping has 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. You can switch a component to this mode by clicking the **File** (

) button, and selecting **Use Dynamic File Names Supplied by Mapping**.



FileNamesAsParameters.mfd (MapForce Enterprise Edition)

If you double-click the title bar of either of the **InputFileName** and **OutputFileName** components, you can view or edit their properties. For example, you can specify the data type of the input parameter or change the input parameter name, as described in Simple Input Component Settings (140). In this example, the input and output parameters are configured as follows:

- The **InputFileName** parameter is of type "string" and it has a default value supplied by a constant defined in the same mapping. The constant is of type "string" and its value is "Altova_Hierarchical.xml". Therefore, when this mapping runs, it will attempt to read data from a file called "Altova_Hierarchical.xml", assuming that you do not supply some other value as parameter.
- The **OutputFileName** parameter is of type "string" and it also has a default value supplied by a constant defined in the same mapping. The constant is of type "string" and its value is "Altova_Hierarchical_output.xml". Therefore, the mapping will create an XML output file called "Altova_Hierarchical_output.xml" when it runs, assuming that you do not supply some other value as parameter.

The following sections illustrate how to run the mapping and supply parameters in the following transformation languages:

- XSLT 2.0 (144), using RaptorXML Server

XSLT 2.0

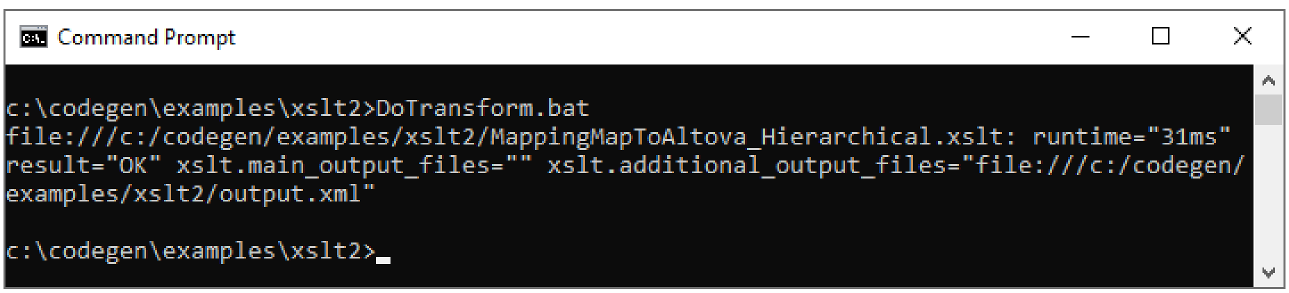
If you generate code in XSLT 1.0, XSLT 2.0, or XSLT 3.0, a **DoTransform.bat** batch file is generated in the chosen target directory, in addition to the XSLT file. The **DoTransform.bat** lets you execute the mapping with RaptorXML Server, see Automation with RaptorXML Server (426).

To use a different input (or output) file, edit the **DoTransform.bat** file to include the required parameters, as follows:

1. First, generate the XSLT code. For example, to generate XSLT 2.0, select the menu command **File | Generate Code In | XSLT 2.0**.
2. Copy the **Altova_Hierarchical.xml** file from **<Documents>\Altova\MapForce2023\MapForceExamples** to the directory where you generated the XSLT 2.0 code (in this example, **c:\codegen\examples\xslt2**). As stated previously, the mapping will attempt to read this file if you do not supply a custom value to the **InputFileName** parameter.
3. Edit **DoTransform.bat** to include the custom input parameter either before or after `%*`. Note that the parameter value is enclosed with single quotes. The available input parameters are listed in the **rem** (Remark) section. Let's suppose that you would like to generate an output file called **output.xml**. To achieve this, change the **DoTransform.bat** file as follows:

```
@echo off

RaptorXML xslt --xslt-version=2
  --input="MappingMapToAltova_Hierarchical.xslt"
  --param=OutputFileName:'output.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. If you followed the steps above, the name of the generated output file will be **output.xml**.



5.2 Simple Output

An output component (or "simple output") is a MapForce component which enables you to return a string value from the mapping. Output components represent one possible type of target components [65], but should not be confused with the latter. 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 [147]).

Simple output components should not be confused with output parameters of user-defined functions (see User-Defined Functions [198]). There are some similarities and differences between the two, as follows.

Output components	Output parameters of user-defined functions
Added from **Function | Insert Output** menu.	Added from **Function | Insert Output** 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:

<table>
<tr><th>Language</th><th>How it works</th></tr>
<tr><td>XSLT 1.0, XSLT 2.0, XSLT 3.0</td><td>
In the generated XSLT files, a simple output component defined in the mapping becomes the output of the XSLT transformation.
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.
To write the output to a file, add or edit the <code>--output</code> parameter in the DoTransform.bat file. For example, the following DoTransform.bat file has been edited to write the mapping output to the Output.txt file (see highlighted text).
<pre>RaptorXML xslt --xslt-version=2 --input="MappingMapToResult1.xslt" --output="Output.txt" %* "MappingMapToResult1.xslt"</pre>
</td></tr>
</table>

Language	How it works
	If an `--output` 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.

5.2.1 Adding Simple Output Components

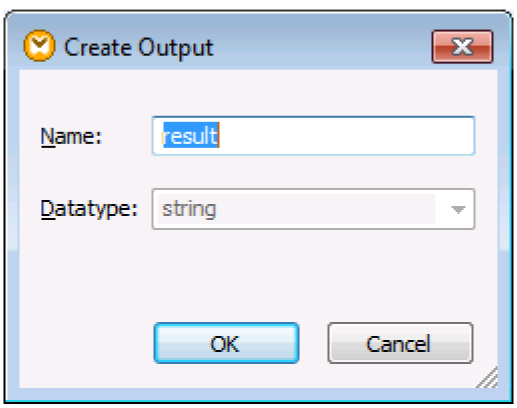
To add an output component to the mapping area:

1. Make sure that the mapping window displays the main mapping (not a user-defined function).
2. Do one of the following:

   a. On the **Function** menu, click **Insert Output**.

   b. Click the **Insert output**  toolbar button.

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**.

5.2.2 Example: Previewing Function Output

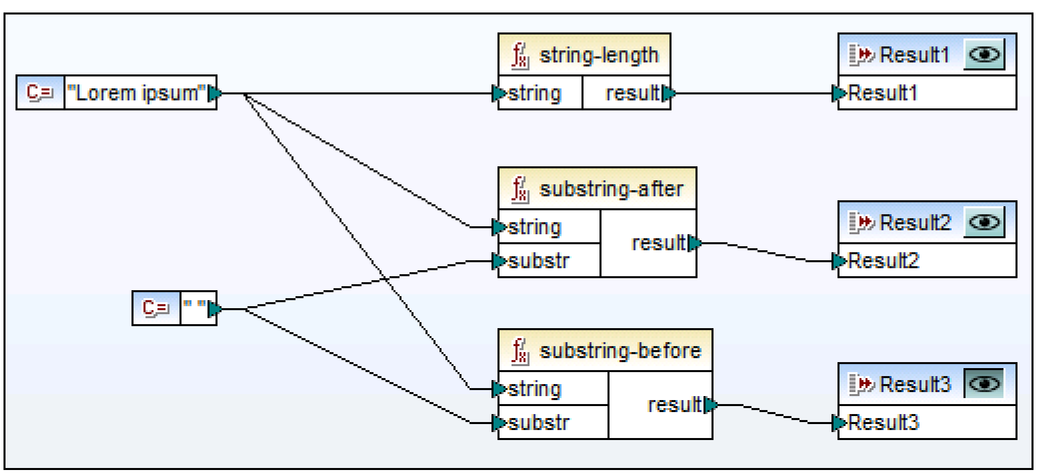
This example illustrates how to preview the output returned by MapForce functions with the help of simple output components. You will make the most of this example if you already have a basic understanding of functions in general, and of MapForce functions in particular. If you are new to MapForce functions, you may want to refer to Using Functions [190] 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:

`string-length` [300]	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.
`substring-after` [301]	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".
`substring-before` [302]	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.

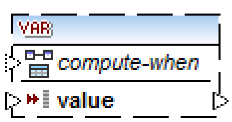
5.3 Variables

A variable is a special type of components used to store an intermediate mapping result for further processing. Variables can be of simple type (e.g., string, integer, boolean, etc) and complex type (a tree structure). See the examples of both types in the subtopics below.

One of the most important aspects of variables is that they are sequences and can be used to create sequences. The term *sequence* means a list of zero or more items. This makes it possible for a variable to process multiple items for the duration of the mapping lifetime. For more information, see also Mapping Rules and Strategies [397]. However, it is also possible to assign a value to a variable once and keep this value the same for the rest of the mapping. For details, see Changing the Context and Scope of Variables [156].

Simple variables

A simple variable is built to represent atomic types such as strings, numbers, and booleans (*see screenshot below*).



Complex variables

A complex variable has a tree structure. The structures on which a complex variable can be based are summarized in the list below.
MapForce Basic Edition:

- XML Schema Structure

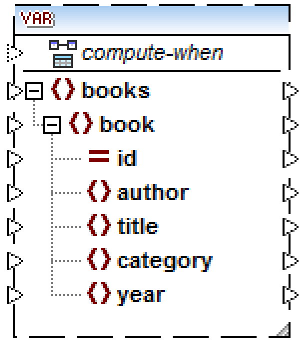
MapForce Professional Edition:

- XML Schema Structure
- Database Structure

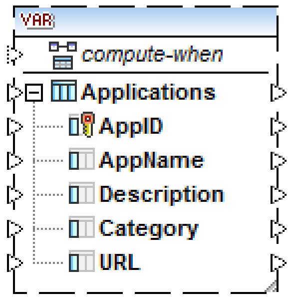
MapForce Enterprise Edition:

- XML Schema Structure
- Database Structure
- EDI Structure
- FlexText Structure
- JSON Schema Structure

Example 1: Variable based on XML Schema
You can create a complex variable by supplying an XML schema which defines the structure of the variable (*see screenshot below*). 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 an associated instance XML file. The data of the variable is computed at mapping runtime.



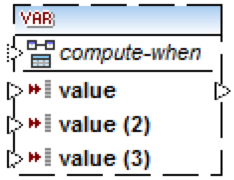
Example 2: Variable based on a database (MapForce Professional and Enterprise editions)
If you choose a database structure for your variable (*see screenshot below*), you can choose a specific database table as the root item for the variable structure. MapForce allows you to create DB-based variables with a tree of related tables. The tree of related tables represents an in-memory structure that has no connection to the database at runtime. This also means that there is no automatic handling of foreign keys and no table actions in parameters or variables.



Compute-when
In both examples above, 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 in the mapping. For more information, see Changing the Context and Scope of Variables 156.

Variables with duplicated inputs

When necessary, items of a variable structure can be duplicated to accept data from more than one source connection. This is similar to duplicating inputs 72 in standard components. This does not apply, however, to variables created from database tables. The screenshot below illustrates a simple variable with duplicated inputs.



Chained mappings vs. variables

Variables can be compared to intermediate components of a chained mapping[371]. However, variables are more flexible and convenient if you do not need to produce intermediary files at each stage of the mapping. The table below outlines differences between variables and chained mappings.

Chained mappings	Variables
Chained mappings involve two independent steps. For example, a mapping has three components, namely A, B, and C. Step 1: mapping data A to B. Step 2: mapping data from B to C.	You can control when and how often the variable value is computed when the mapping is carried out. For details, see Changing the Context and Scope of Variables[156].
When the mapping is carried out, intermediate results are stored externally in files.	When the mapping is carried out, intermediate results are stored internally. No external files containing the results of a variable are produced.
The intermediate result can be previewed using the preview button.	The result of a variable cannot be previewed, since it is computed at mapping runtime.

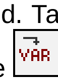
Note: Variables are not supported if the mapping transformation language is set to XSLT 1.0.

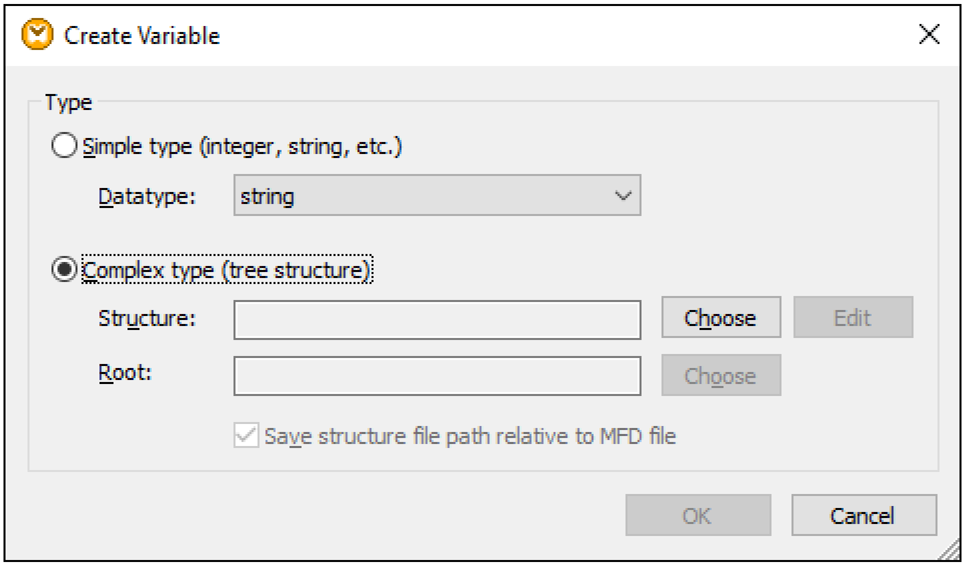
5.3.1 Add a Variable

This topic explains how to add a variable to a mapping. The first option is to add a variable via the menu or toolbar command. The second option allows you to add a variable via the context menu.

Option 1: via the menu or toolbar command

This option enables you to add a variable via the menu or toolbar command. Take the steps below:

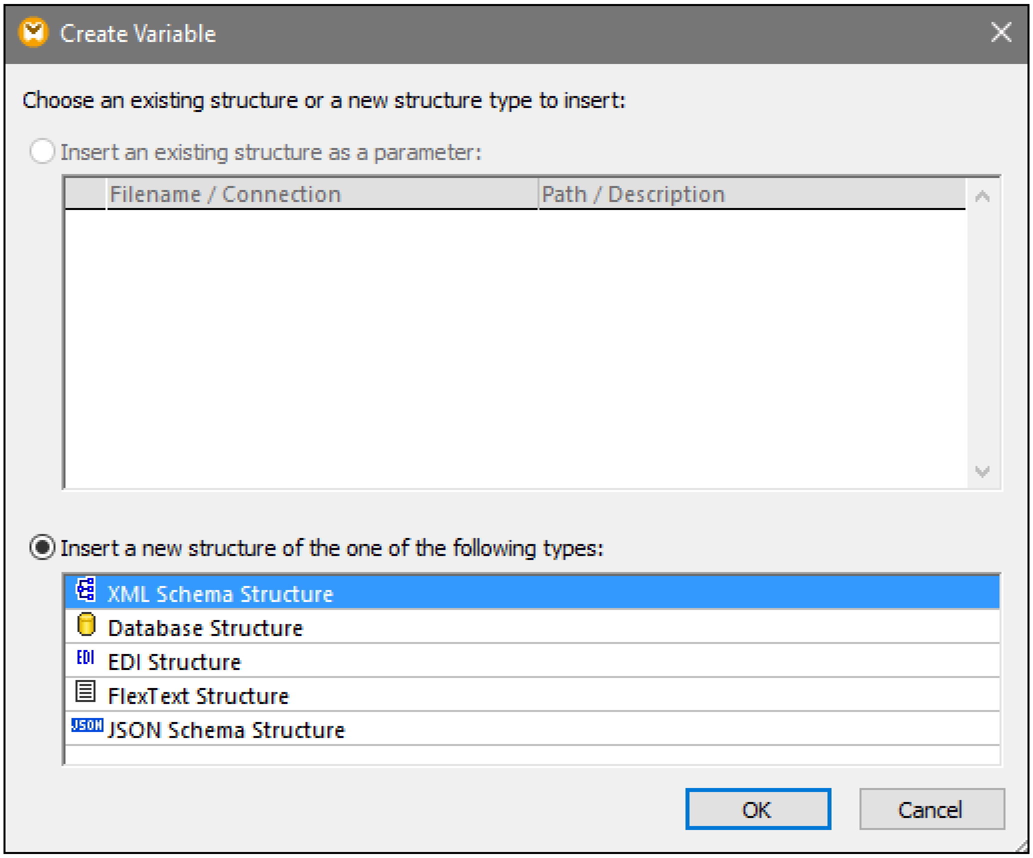
1. Go to the **Insert** menu and click **Variable**. Alternatively, click the  toolbar button (**Variable**).



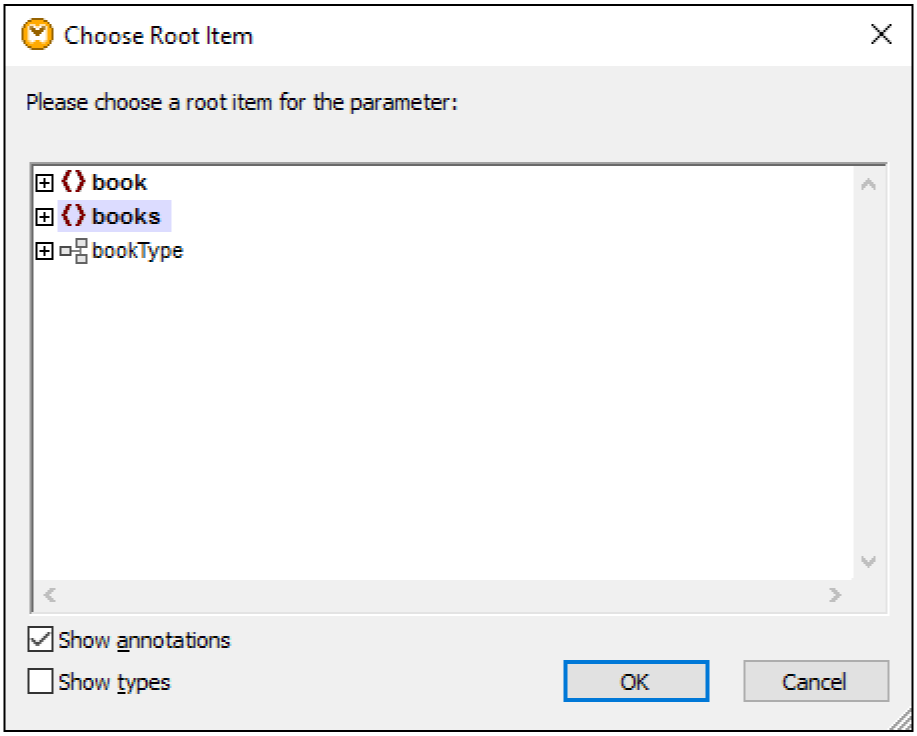
2. Select the type of variable you want to insert (simple or complex type).

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 150. The structures illustrated in the screenshot below only apply to MapForce Enterprise Edition. See the list of structures relevant to other MapForce editions in the previous topic 150.



4. When prompted, specify the root item of the structure of the variable. For example, in XML schemas, you can select any element or type from the selected source (*see screenshot below*).

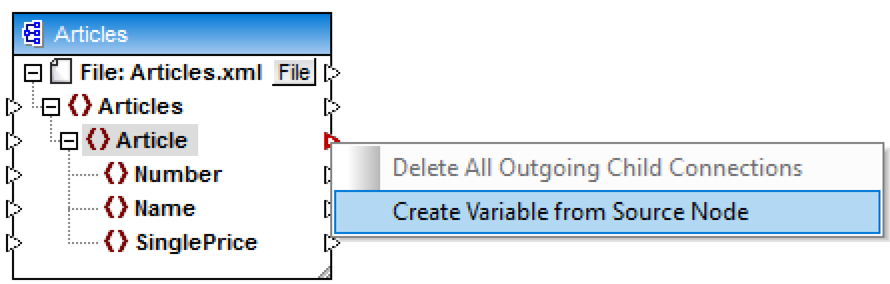


Option 2: via the context menu

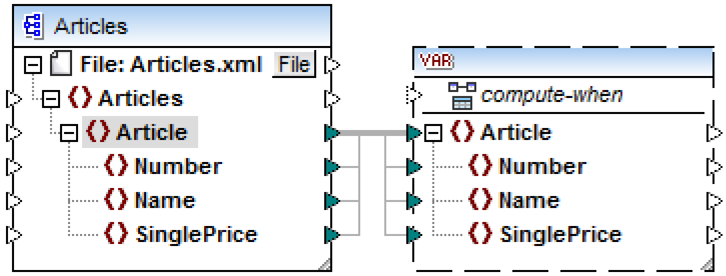
The second option allows you to create a variable using the context menu. The possible options are listed below.

Variable from a source node

To create a variable from a source node, right-click the output connector of a component (in this example, the output connector of the `<Article>` element) and select **Create Variable from Source Node** (*see screenshot below*).



This creates a complex variable with the source schema of the `Articles` component. All the items are automatically connected with a copy-all connection [86] (*see screenshot below*).

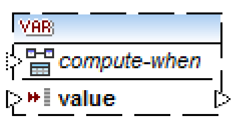


Variable from a target node
To create a variable from a target node, right-click the input connector of a target component and select **Create Variable for Target Node**. This creates a complex variable with the same schema as in the target. All the items are automatically connected with a copy-all connection.

Variable from a filter:
To create a variable using a filter, 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 with the source schema and automatically uses the item linked to the filter input as the root element of the intermediate component.

5.3.2 Scope and Context of Variables

Every variable has a `compute-when` input item (*see screenshot below*), which allows you to control the scope of the variable. This means that you can control 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 following terms are relevant to the discussion of the scope and context of variables: *subtree* and *variable value*. A subtree is a 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 is 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* 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 201 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.

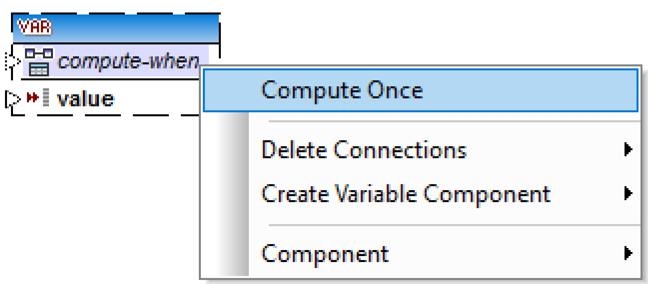
Compute-when is connected

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 is related to the general rule governing connections in MapForce: For each source item, one target item is created. In this case, `compute-when` instructs MapForce to compute the variable value for each source item. For more information, see Mapping Rules and Strategies 397.

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:



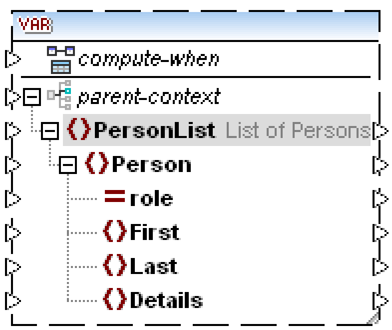
When you change the scope of a variable to `compute-when=once`, the input connector is removed from the `compute-when` item, since such a variable is only evaluated once. In a user-defined function, the `compute-when=once` variable is evaluated each time the function is called before the actual function result is evaluated.

Parent-context

The `parent-context` argument is an optional argument in some MapForce core aggregation functions (e.g.,

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.

Adding a `parent-context` item may be necessary, for example, if your mapping uses multiple filters and you need an additional parent node to iterate over. For details, see Example: Changing the Parent Context 404. 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.

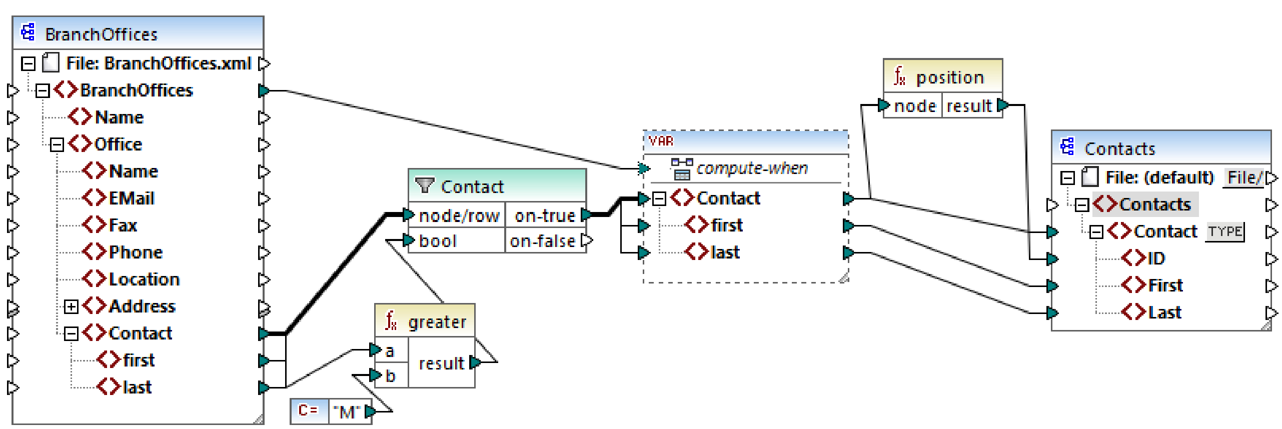


The parent context adds a virtual parent node to the hierarchy within the component. This allows you to iterate over an additional node in the same or in a different source component.

5.3.3 Example: Filtering and Numbering Nodes

The mapping illustrated in this example is available as **PositionInFilteredSequence.mfd** in the **<Documents>\Altova\MapForce2023\MapForceExamples** folder.

This mapping reads an XML file which contains contact data of several people, filters them, and writes them to a target XML file. The goal of the mapping is to filter from the source XML file only those people whose last name begins with letter "M" or a subsequent letter. Secondly, the extracted contacts must be numbered. The number is going to act as the unique identifier of each contact in the target XML file.



PositionInFilteredSequence.mfd

To achieve the goal above, the following component types were added to the mapping:

- A filter (see Filters and Conditions [168])
- A complex variable (see Adding Variables [152])
- The functions **greater** [253] and **position** [288] (see Add a Function to the Mapping [191])
- A constant (To add a constant, select the menu command **Insert | Constant**).

The variable uses the same schema as the source component. If you right-click the variable and select **Properties** from the context menu, notice that the node **BranchOffices/Office/Contact** is selected as root node for this variable structure.

First, data of the source component is passed on to the filter. The filter passes onwards to the variable only those records that meet the filter condition. Namely, the filter is configured to get only those `Contact` nodes where the last name is equal or greater than M". To achieve this, the function **greater** [253] compares each `last` item with the constant value "M".

The variable has the `compute-when` input connected to the root item of the source component (`BranchOffices`). At runtime, this causes the variable to be re-evaluated whenever a new item is read from the sequence in the source component. In this mapping, however, connecting or not connecting the `compute-when` item does not make a difference. The reason is that the variable is connected to the `Contact` source item (indirectly through the filter), and it would compute as many times as there are instances of `Contact` which meet the filter condition.

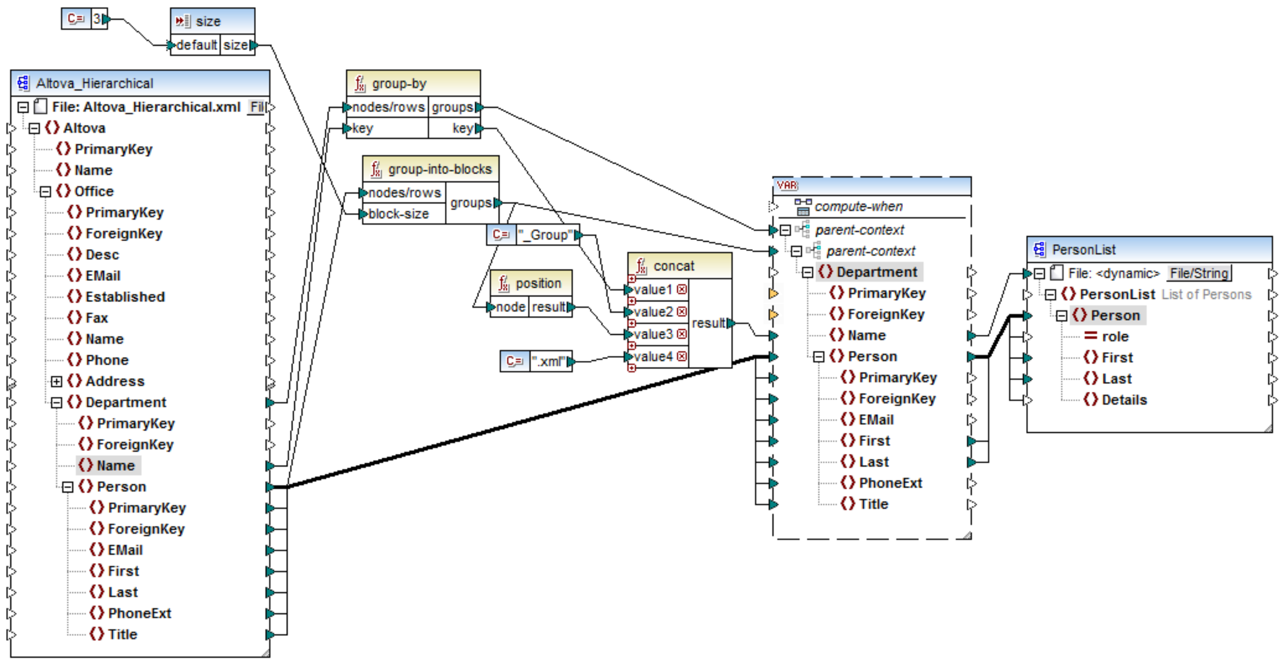
The **position** [288] functions returns, for each iteration of the variable, the number of the current sequence. Only eight contacts meet the filter condition; therefore, if you preview the mapping and look at the output, notice how IDs 1 through 8 were written to the `ID` element of the target component.

In case you were wondering why the variable was necessary at all, it is because of the requirement to number all records. Had we connected the filter result directly to the target component, there would have been no way to number each occurrence of `Contact`. The purpose of the variable in this mapping is, therefore, to store each instance of `Contact` temporarily on the mapping, so that it can be numbered before it is written to the target.

5.3.4 Example: Grouping and Subgrouping Records

The mapping illustrated in this example is available as **DividePersonsByDepartmentIntoGroups.mfd** in the **<Documents>\Altova\MapForce2023\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 [156]).

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`** [277] 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`** [280] 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 [139].

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`** [296] 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"

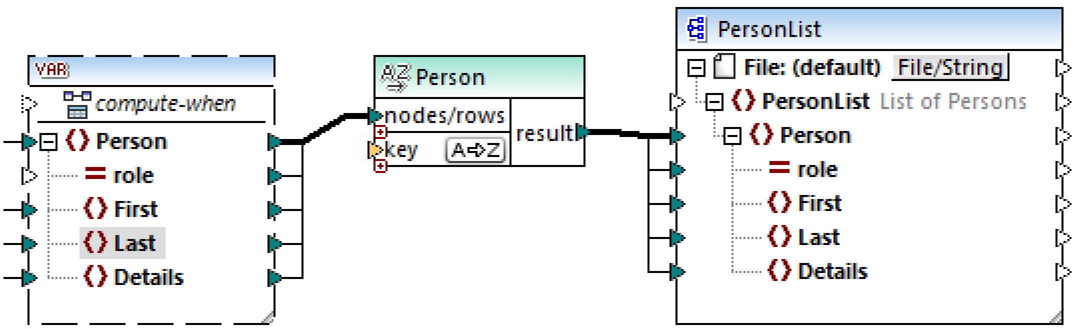
The result of this concatenation is stored in the `Name` item of the variable, and then supplied as a dynamic file name to the target component. This causes a new file name to be created for each received value. In this example, the variable computes eight groups in total, so eight output files are created when the mapping runs, as required. For more information about this technique, see Processing Multiple Input or Output Files Dynamically 417.

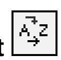
5.4 Sort Components

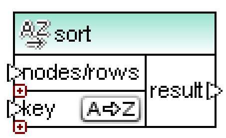
To sort input data based on a specific sort key, use a Sort component. The Sort component supports the following target languages: XSLT2, XQuery, and Built-in.

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).



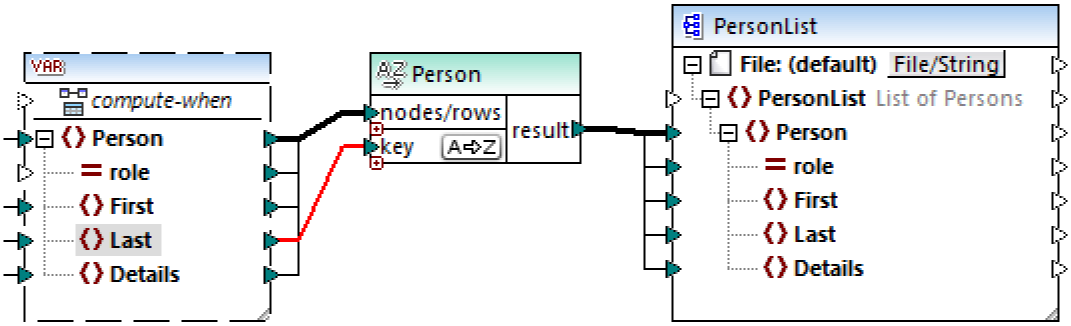
- On the **Insert** menu, click **Sort** (alternatively, click the **Sort**  toolbar button). This inserts the Sort component in its "unconnected" form.



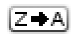
As soon as a connection is made to the source component, the title bar name changes to that of the item connected to the `nodes/rows` item.

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`.

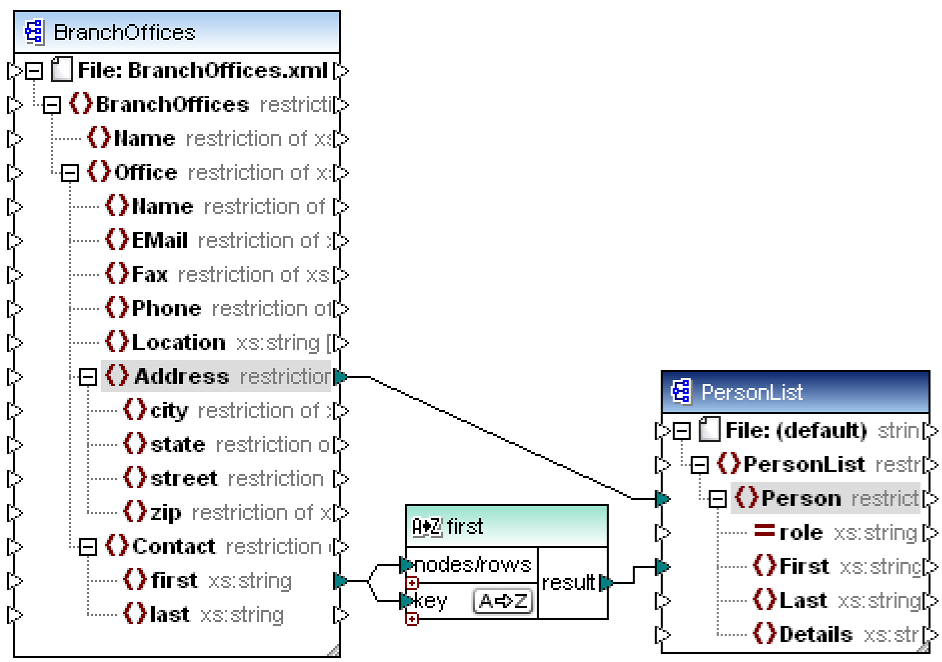


To change the sort order:

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

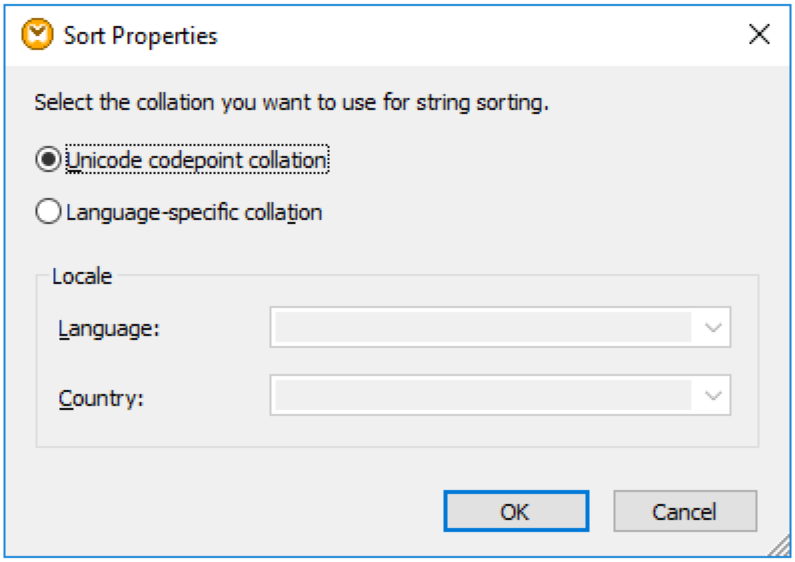
To sort input data consisting of simple type items:

- 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.

5.4.1 Sorting by Multiple Keys

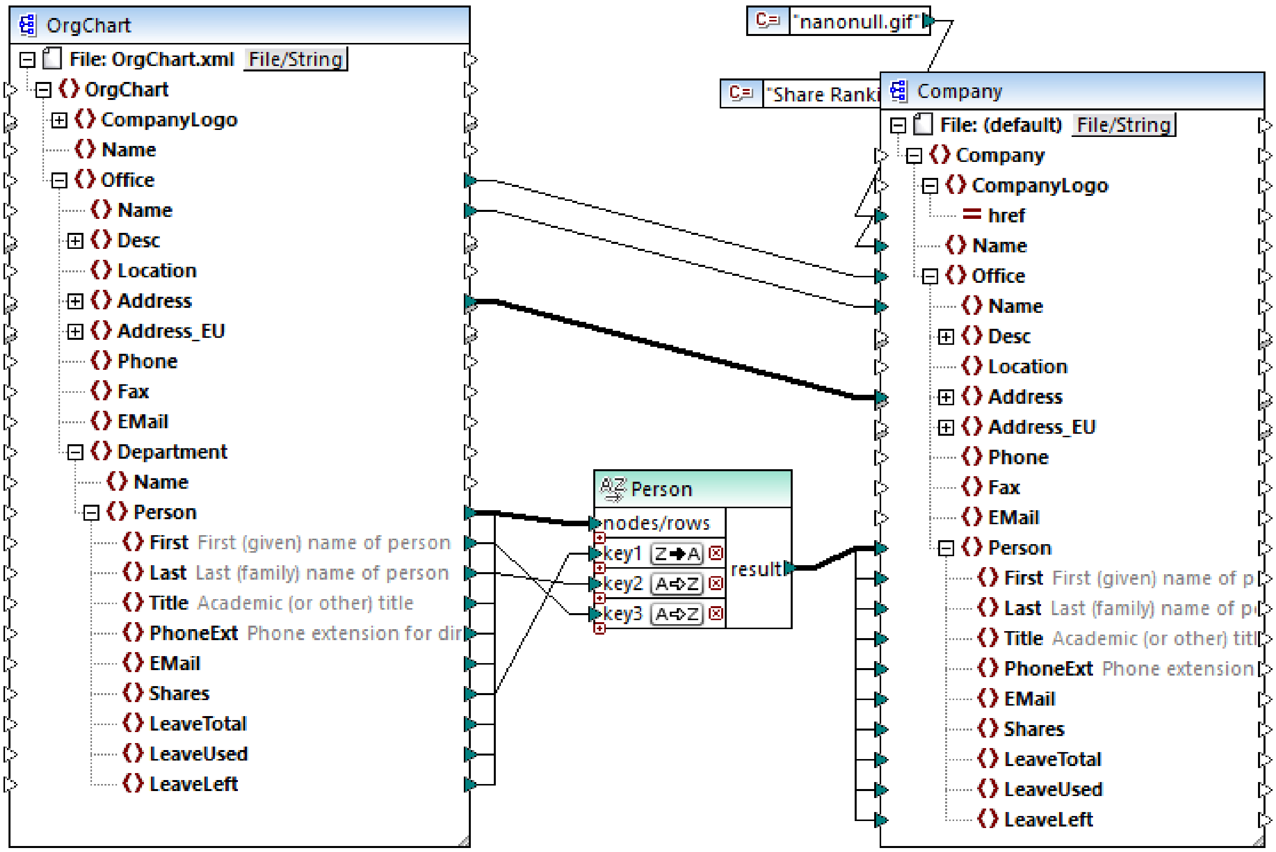
After you add a Sort component to the mapping, one sorting key called `key` is created by default.

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\MapForce2023\MapForceExamples\SortByMultipleKeys.mfd**.



SortByMultipleKeys.mfd

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

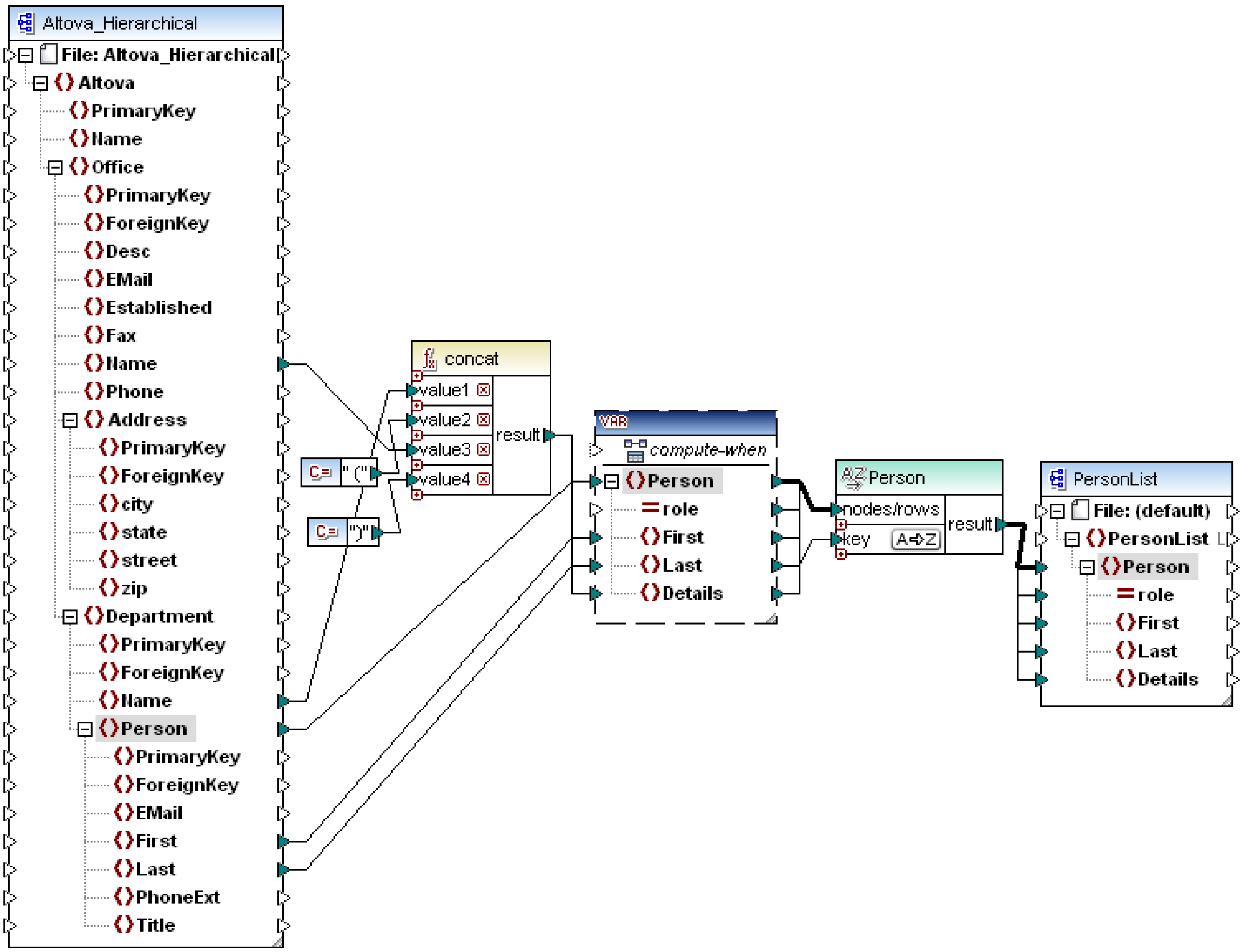
1. `Shares` (number of shares a person holds)
2. `Last` (last name)
3. `First` (first name)

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).

5.4.2 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\MapForce2023\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 [Add a Function to the Mapping](191).
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 [397]). 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 [150] and Copy-All Connections [86].

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.

5.5 Filters and Conditions

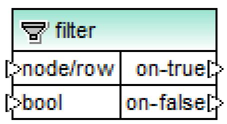
When you need to filter data, or get a value conditionally, you can use one of the following component types:

- 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 169.

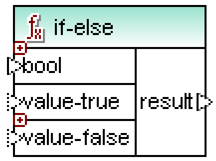
Returning a value conditionally

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>
```

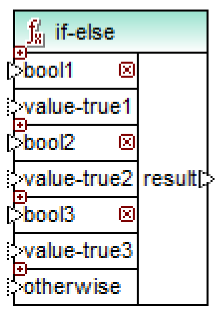
An **If-Else Condition** would enable you to return, for each item in the list, the value "high" if temperature exceeds 20 degrees Celsius, and value "low" if temperature is lower than 5 degrees Celsius.

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.



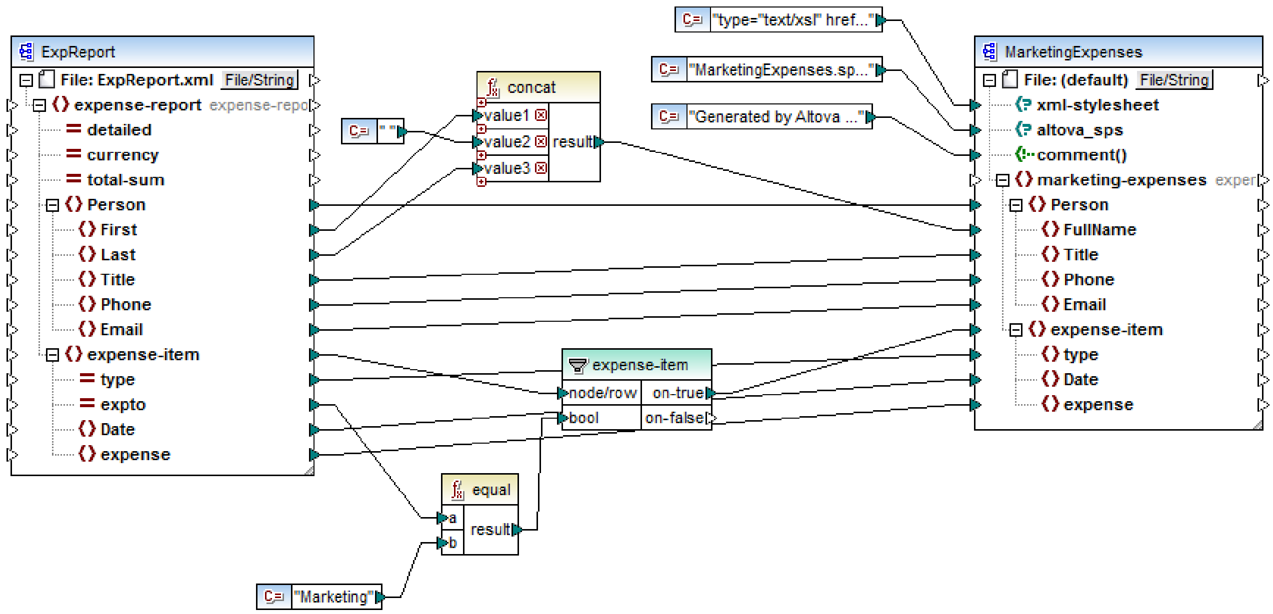
Expanded **If-Else Conditions** are evaluated from top to bottom (first conditions is checked first, then the second one, and so on). If you want to return a value when none of the conditions are true, connect it to **otherwise**.

For a step-by-step mapping example, see Example: Returning a Value Conditionally 171.

5.5.1 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\MapForce2023\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, see also Add a Function to the Mapping[191]. The **equal** function compares the value of the `expto` 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:

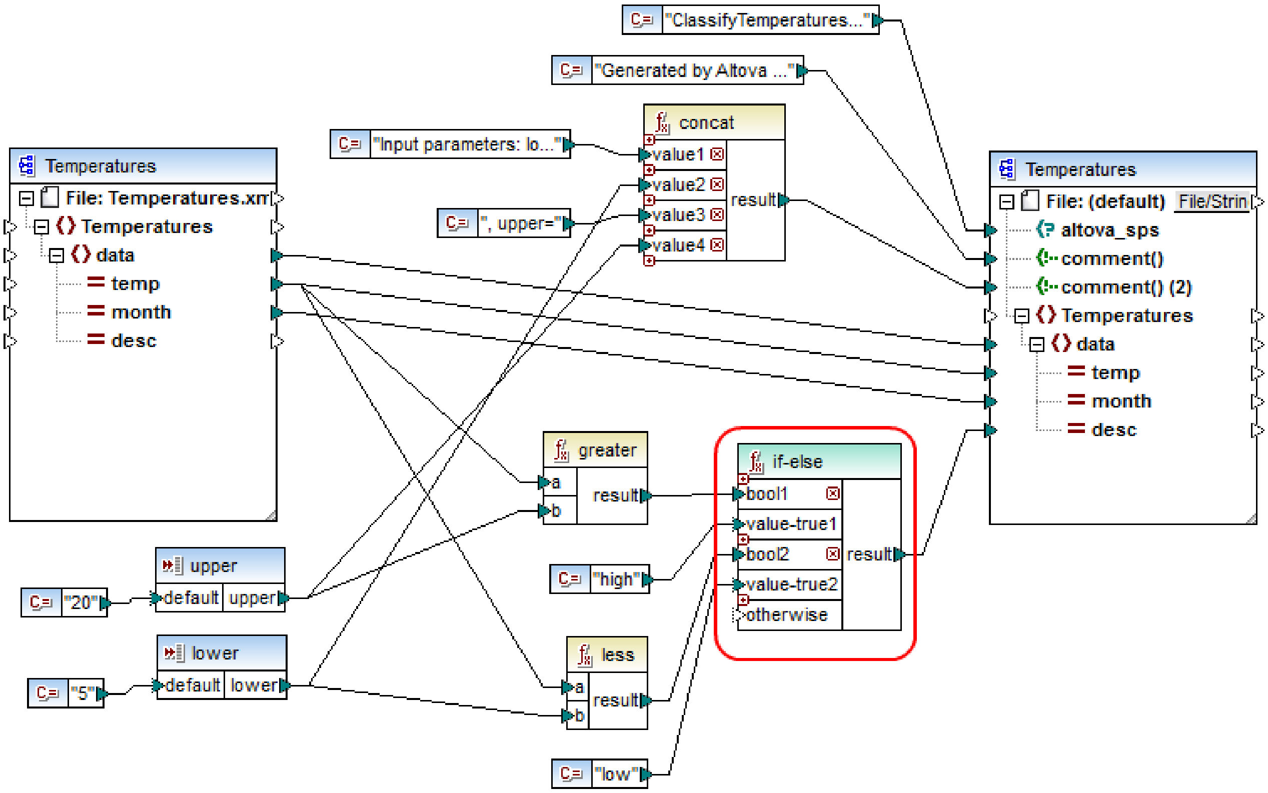
```
...
   <expense-item>
      <type>Meal</type>
      <Date>2003-01-01</Date>
      <expense>122.11</expense>
   </expense-item>
   <expense-item>
      <type>Lodging</type>
      <Date>2003-01-02</Date>
      <expense>299.45</expense>
   </expense-item>
...
```

XML output after the mapping is executed

5.5.2 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[169]).

The mapping described in this example is available at the following path: **<Documents>\Altova\MapForce2023\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, see also Add a Function to the Mapping 191. 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 139.)

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.

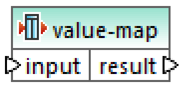
When you are ready to preview the mapping result, click the **Output** tab. Notice that the resulting XML output now includes the "desc" attribute, whenever the temperature is either greater than 20 or lower than 5.

```
...
   <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

5.6 Value-Maps

The Value-Map component enables you to replace a value by another value with the help of a predefined look-up table. Such a component processes only one value at a time; therefore, it has one **input** and one **result** on the mapping.



A Value-Map is very useful when you would like to map individual items within two sets in order to replace items. For example, you could map the days of the week expressed as numbers (1, 2, 3, 4, 5, 6, and 7) to the name of each day of the week ("Monday", "Tuesday", and so on). Likewise, you could map the month names ("January", "February", "March", etc) to the numeric representation of each month (1, 2, 3, etc). At mapping run time, the matching values will be replaced according to your custom look-up table. The values in both sets can be of different type, but each set must store values of the same data type.

Value-Map components are suitable for simple look-ups, where each value in the first set corresponds to a single value in the second set. If a value is not found in the look-up table, you can either replace it with a custom value or an empty value, or pass it on as is. If you need to look up or filter values based on more complex criteria, use one of the filtering components 168 instead.

Importantly, when you generate code or compile a MapForce Server Execution file from the mapping, the look-up table data is embedded into the generated code or file. Consequently, defining a look-up table directly on the mapping is a good choice only if your data does not change frequently and is not very big (less than maybe a few hundred entries). If the look-up data changes regularly, you may find it difficult to maintain both the mapping and the generated code regularly—it is easier to maintain the look-up data as text, XML, database, or perhaps Excel.

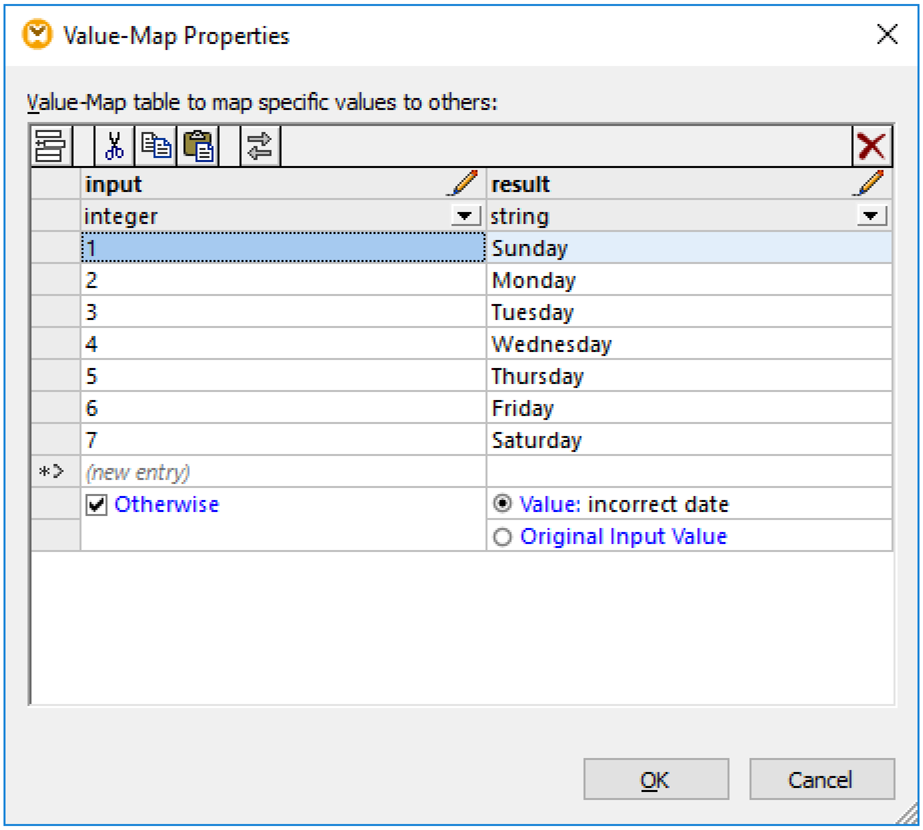
If the look-up table is huge, the mapping execution will be slowed down by the look-up table. In this case, it is recommended to use a database component with SQL-Where instead. Database components are available in MapForce Professional and Enterprise editions. SQLite databases are good candidates for this, given their portability. On the server side, you can improve the performance of look-up tables by running a mapping with MapForce Server or MapForce Server Advanced Edition.

Creating Value-Maps

To add a Value-Map component to the mapping, do one of the following:

- Click the **Insert Value-Map**  toolbar button.
- On the **Insert** menu, click **Value-Map**.
- Right-click a connection, and select **Insert Value-Map** from the context menu.

This adds a new Value-Map component to the mapping. You can now start adding pairs of items to the look-up table. To do this, double-click the component's title bar or right-click it and select **Properties** from the context menu.



At mapping run time, MapForce checks each value that reaches the **input** of the Value-Map. If there is a matching value *in the left column* of the look-up table, then it replaces the original input value with the value *from the right column*. Otherwise, you can optionally configure it to return one of the following:

- A replacement value. In the example above, the replacement value is the text "incorrect date". You can also set the replacement value to be empty, by not entering any text at all.
- The original input value. This means that, if no match is found in the look-up table, the original input value will be passed further on to the mapping, unchanged.

> If you do not configure an "Otherwise" condition, the Value-Map returns an **empty node** whenever a match is not found. In this case, nothing will be passed to the target component and the output will contain missing fields. To prevent this from happening, you should either configure the "Otherwise" condition, or use the **`substitute-missing`**[295] function.

There is a difference between setting an empty replacement value and not specifying the "Otherwise" condition. In the first case, the field will be generated in the output, but it will have an empty value. In the latter case, the field (or XML element) enclosing the value will not be created at all. For more information, see Example: Replacing Job Titles[180].

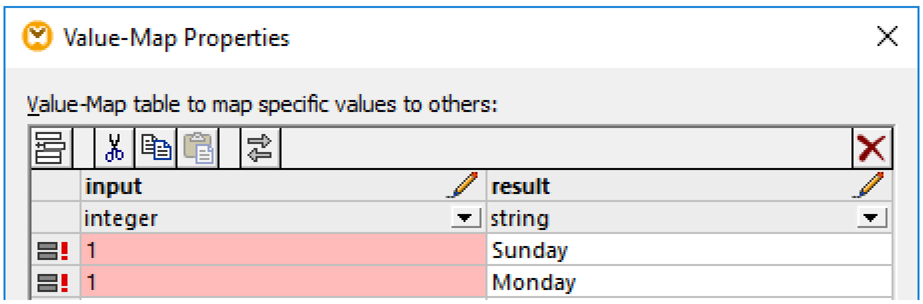
Populating a Value-Map

In a look-up table, you can define as many pairs of values as needed. You can enter the values manually, or copy-paste tabular data from text, CSV, or Excel files. Copy-pasting tables from an HTML page using a common browser will also work in most cases. If you copy data from text files, the fields must be separated by tab characters. In addition, MapForce will recognize text separated by commas or semicolons in most cases.

Keep in mind the following when creating look-up tables:

1. All items in the left column must be unique. Otherwise, it would not be possible to determine which item you want to match specifically.
2. Items that belong to the same column must be of the same data type. You can choose the data type from the drop-down list at the top of each column in the look-up table. If you need to convert Boolean types, enter the text "true" or "false" literally. For an illustration of this case, see Example: Replacing Weekdays[177].

If MapForce encounters invalid data in the look-up table, it displays an error message and highlights the invalid rows in pink color, for example:



To import data from an external source into the Value-Map component:

1. Select the cells of interest in the source program (for example, Excel). This can be either a single column of data or two adjacent columns.
2. Copy data to clipboard using the **Copy** command of the external program.
3. On the Value-Map component, click the row before which you would like to paste the data .
4. Click the **Paste table from clipboard** button on the Value-Map component. Alternatively, press **Ctrl+V** or **Shift+Insert**.

Note: The **Paste table from clipboard** button is enabled only if you have copied data from some source first (that is, if there is data on the clipboard).

When your clipboard data contains multiple columns, then only data from the first two columns are inserted into the look-up table; any other subsequent columns will be ignored. If you paste data from a single column on top of any existing values, a context menu appears, asking whether the clipboard data should be inserted as new rows or the existing rows should be overwritten. Therefore, if you need to overwrite existing values in the look-up table as opposed to inserting new rows, ensure that the clipboard contains only one column, not multiple.

To insert rows manually before an existing row, first click the row of interest, and then click the **Insert** button.

To move an existing row to some other position, drag the row to the new position (upwards or downwards) while holding the left mouse button pressed.

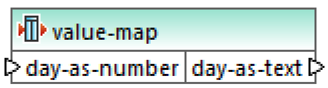
To copy or cut rows for subsequent pasting at some other position, first select the row, and then click the **Copy** button (or **Cut** button, respectively). You can also copy or cut multiple rows that are not necessarily consecutive. To select multiple rows, hold the **Ctrl** key pressed while clicking the rows. Note that the cut or copied text always contains values from both columns; you cannot cut or copy values from one column only.

To remove a row, click it, and then click the **Delete** button.

To swap the left and right columns, click the **Swap** button.

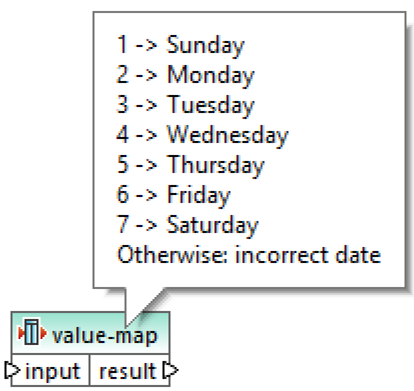
Renaming Value-Map parameters

By default, the input parameter of a Value-Map component is called "input" and the output parameter is called "result". To make the mapping clearer, you can optionally rename any of these parameters by clicking the **Edit** button next to the respective name. The following is an example of a Value-Map with custom parameter names:



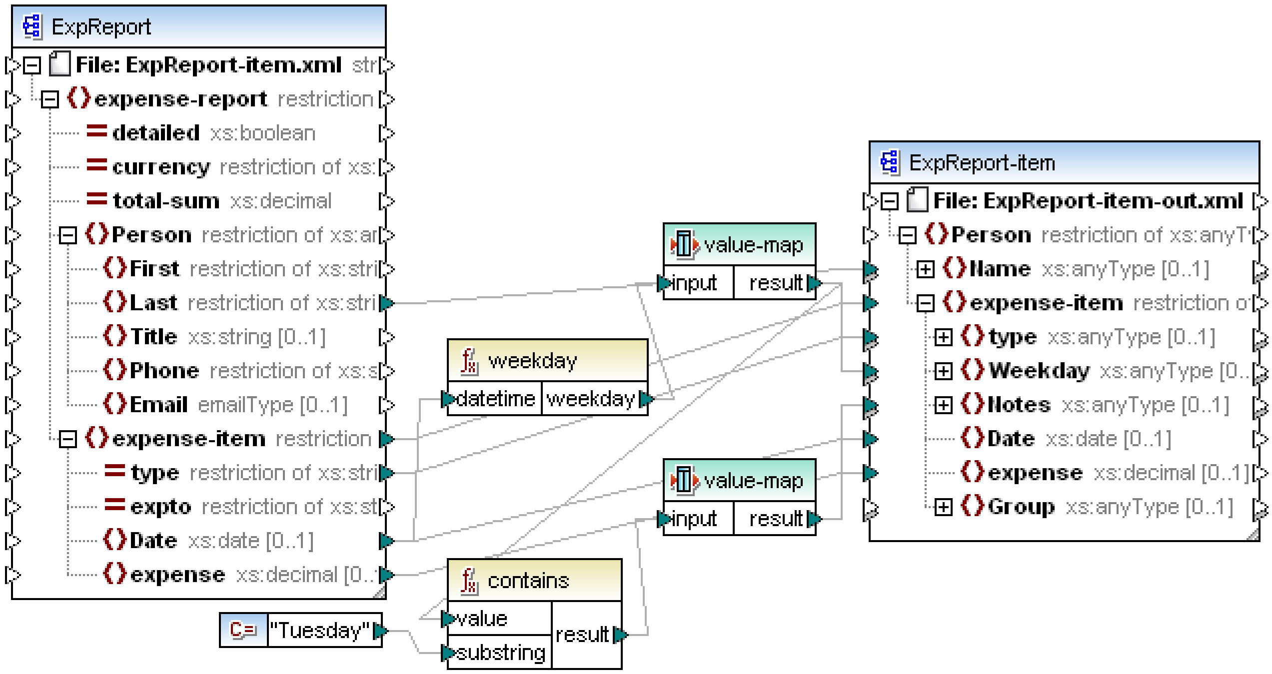
Previewing a Value-Map

After you have finished creating a Value-Map, you can quickly preview its implementation directly from the mapping by holding the mouse over the component's title bar:



5.6.1 Example: Replacing Weekdays

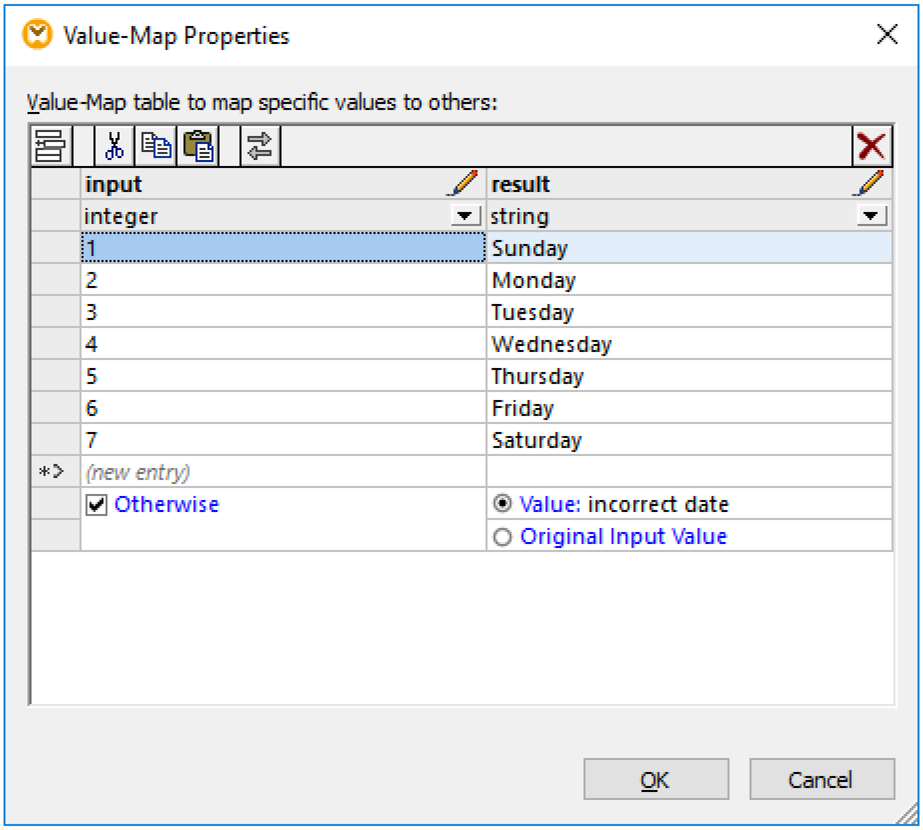
This example illustrates a Value-Map that replaces integer values with weekday names (1 = Sunday, 2 = Monday, and so on). This example is accompanied by a mapping which is available at the following path: **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\Expense-valmap.mfd**.



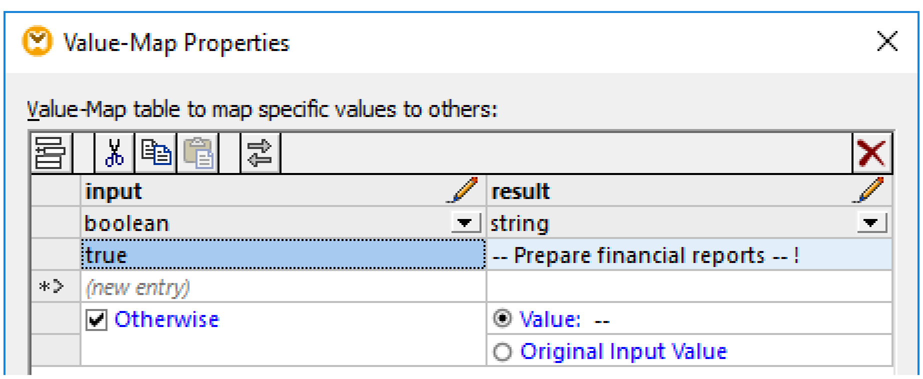
Expense-valmap.mfd

This mapping extracts the day of the week from the **Date** item in the source file, converts the numerical value into text, and writes it to the **Weekday** item of the target component. More specifically, the following happens:

- The **weekday** function extracts the weekday number from the **Date** item in the source file. The result of this function are integers ranging from 1 to 7.
- The first Value-Map component transforms the integers into weekdays (1 = Sunday, 2 = Monday, and so on). If the component encounters an invalid integer outside of the 1-7 range, then it will return the text "incorrect date".



- If the weekday contains "Tuesday", then the text "Prepare Financial Reports" is written to the **Notes** item in the target component. This is achieved with the help of the **contains** function, which passes a Boolean **true** or **false** value to a second Value-Map component. The second Value-Map has the following configuration:



The Value-Map illustrated above should be understood as follows:

- Whenever a Boolean **true** is encountered, convert it to the text "-- Prepare financial reports -- ! ". For all other cases, return the text "--".

Notice that the data type of the first column is set to "boolean". This ensures that the input Boolean value **true** is recognized as such.

5.6.2 Example: Replacing Job Titles

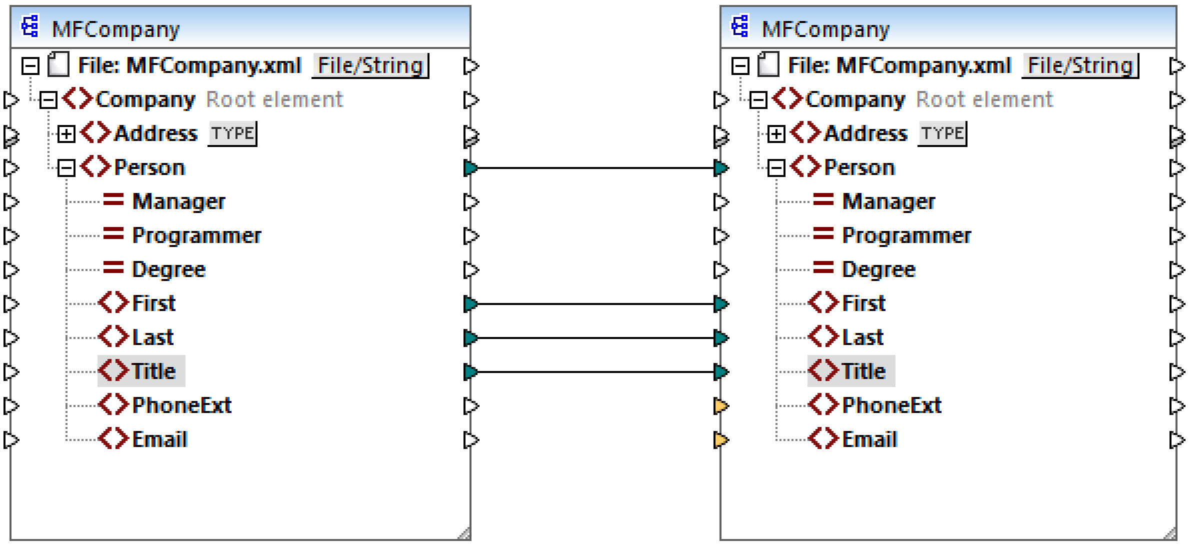
This example shows you how to replace values of specific elements in an XML file with the help of Value-Map components (that is, using a predefined look-up table).

The XML file required for this example is available at the following path: **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\MFCompany.xml**. It stores, among other data, information about company employees and their job titles, for example:

```xml
<Person>
   <First>Michelle</First>
   <Last>Butler</Last>
   <Title>Software Engineer</Title>
</Person>
<Person>
   <First>Lui</First>
   <Last>King</Last>
   <Title>Support Engineer</Title>
</Person>
<Person>
   <First>Steve</First>
   <Last>Meier</Last>
   <Title>Office Manager</Title>
</Person>
```

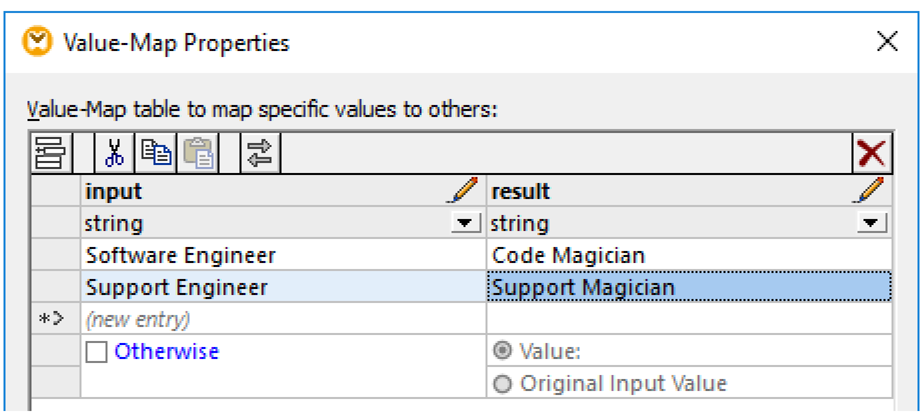
Let's assume that you need to replace some of the job titles in the XML file above. Specifically, the title "Software Engineer" must be replaced with "Code Magician". Also, the title "Support Engineer" must be replaced with "Support Magician". All the other job titles must remain unchanged.

To achieve the goal, add the XML file to the mapping area, by clicking the **Insert XML Schema/File** toolbar button or by running the **Insert | XML Schema/File** menu command. Next, copy-paste the XML component on the mapping and create the connections as shown below. Note that you might need to turn off the **Toggle auto-connect of children** toolbar option first, in order to prevent unnecessary connections from being created automatically.



The mapping created so far simply copies the **Person** elements to the target XML file, without making any changes to the **First**, **Last**, and **Title** elements.

To replace the required job titles, let's add a Value-Map component. Right-click the connection between the two **Title** elements, and select **Insert Value-Map** from the context menu. Set up the Value-Map properties as shown below:

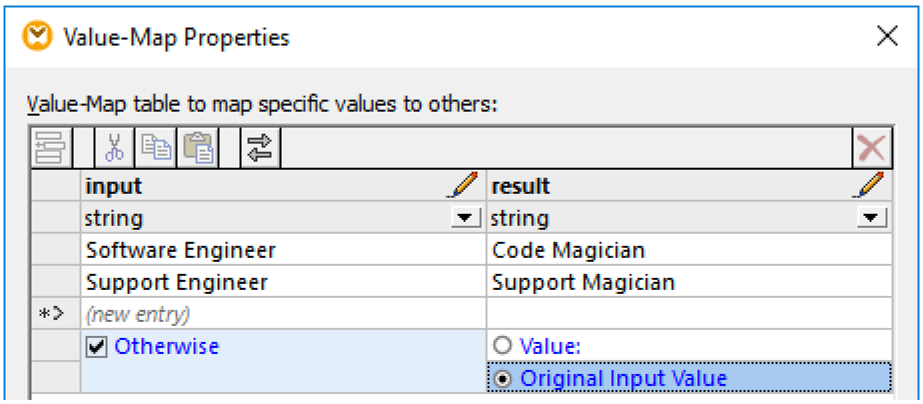


According to the setup above, each occurrence of "Software Engineer" will be replaced with "Code Magician", and each occurrence of "Support Engineer" will be replaced with "Support Magician". Notice that the **Otherwise** condition was not specified yet. For this reason, the Value-Map returns an *empty node* whenever the job title is other than "Software Engineer" and "Support Engineer". Consequently, if you click the **Output** tab and preview the mapping, some of the **Person** elements will have a missing a **Title**, for example:

```
<Person>
   <First>Vernon</First>
   <Last>Callaby</Last>
</Person>
<Person>
```

```
    <First>Frank</First>
    <Last>Further</Last>
</Person>
<Person>
    <First>Michelle</First>
    <Last>Butler</Last>
    <Title>Code Magician</Title>
</Person>
```

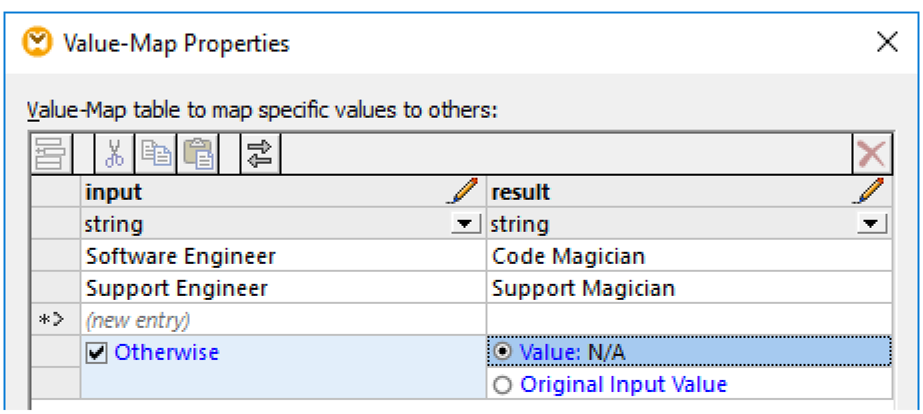
As stated before, empty nodes cause missing entries in the generated output; therefore, in the XML fragment above, only Michelle Butler had the title replaced, because her title was present in the look-up table. The configuration created so far still does not fulfill the original requirement. The correct setup is as follows:



With the configuration above, the following happens at mapping run time:

- Each occurrence of "Software Engineer" will be replaced with "Code Magician"
- Each occurrence of "Support Engineer" will be replaced with "Support Magician"
- If the original title is not found in the look-up table, the Value-Map will return it unchanged.

For illustrative purposes only, we can also change all the job titles other than "Software Engineer" and "Support Engineer" to a custom value, for example "N/A". To achieve this, set the Value-Map properties as shown below:



When you preview the mapping this time, each job title is present in the output, but those that were not matched have the "N/A" value, for example:

```
<Person>
   <First>Vernon</First>
   <Last>Callaby</Last>
   <Title>N/A</Title>
</Person>
<Person>
   <First>Frank</First>
   <Last>Further</Last>
   <Title>N/A</Title>
</Person>
<Person>
   <First>Michelle</First>
   <Last>Butler</Last>
   <Title>Code Magician</Title>
</Person>
```

This concludes the Value-Map example. By applying the logic above, you can now achieve the desired result in other mappings.

5.7 Group Functions

When your mapping must group nodes or rows, you can achieve this with the help of the following MapForce built-in functions:

- `group-by`
- `group-adjacent`
- `group-into-blocks`
- `group-starting-with`
- `group-ending-with`

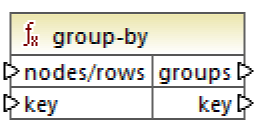
To use any of these functions on the mapping, drag them from the Libraries window onto the mapping area. See also Add a Function to the Mapping 191.

Note: Grouping functions are available in the following languages: XSLT 2.0, XSLT 3.0, C++, C#, Java, Built-In.

The following sections provide typical examples of use for grouping functions. These examples are accompanied by the following demo mapping: **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\GroupingFunctions.mfd**. Note that the demo mapping contains multiple transformations, one for each function. Since only one output can be previewed at a time, remember to click the **Preview** button applicable to the desired transformation before clicking the **Output** tab.

group-by

The `group-by` function creates groups of records according to some grouping key that you specify.



For example, in the abstract transformation illustrated below, the grouping key is "Department". Since there are three unique departments in total, applying the group-by function would create three groups:

Department	Name
Administration	Vernon Callaby
Marketing	Susi Sanna
Engineering	Michelle Butler
Engineering	Fred Landis
Administration	Frank Further

Department	Name
Administration	Vernon Callaby
Administration	Frank Further

Department	Name
Marketing	Susi Sanna

Department	Name
Engineering	Michelle Butler
Engineering	Fred Landis

For more information, see the reference to the group-by [277] function.

group-adjacent

The **group-adjacent** function requires a grouping key as argument, similar to **group-by** function. Unlike **group-by**, this function creates a new group whenever the next key is different. If two adjacent records have the same key, they will be placed into the same group.

For example, in the abstract transformation illustrated below, the grouping key is "Department". The left side of the diagram shows the input data while the right side shows the output data after grouping. The following takes place when the transformation runs:

- Initially, the first key, "Administration", creates a new group.
- The next key is different, so a second group is created, "Marketing".
- The third key is also different, so another group is created, "Engineering".
- The fourth key is the same as the third; therefore, this record is placed in the already existing group.
- Finally, the fifth key is different from the fourth, and this creates the last group.

As illustrated below, "Michelle Butler" and "Fred Landis" were grouped together because they have the same key and are adjacent. However, "Vernon Callaby" and "Frank Further" are in separate groups because they are not adjacent, even though they have the same key.

Department	Name
Administration	Vernon Callaby
Marketing	Susi Sanna
Engineering	Michelle Butler
Engineering	Fred Landis
Administration	Frank Further

Department	Name
Administration	Vernon Callaby

Department	Name
Marketing	Susi Sanna

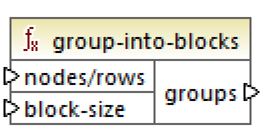
Department	Name
Engineering	Michelle Butler
Engineering	Fred Landis

Department	Name
Administration	Frank Further

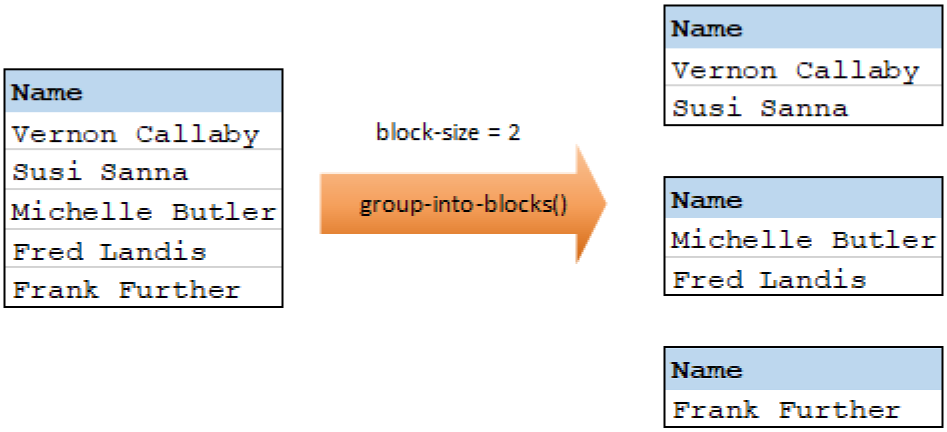
For more information, see the reference to the group-adjacent [275] function.

group-into-blocks

The **group-into-blocks** function creates equal groups that contain exactly N items, where N is the value you supply to the `block-size` argument. Note that the last group may contain N items or less, depending on the number of items in the source.



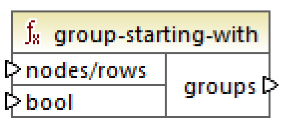
In the example below, `block-size` is 2. Since there are five items in total, each group contains exactly two items, except for the last one.



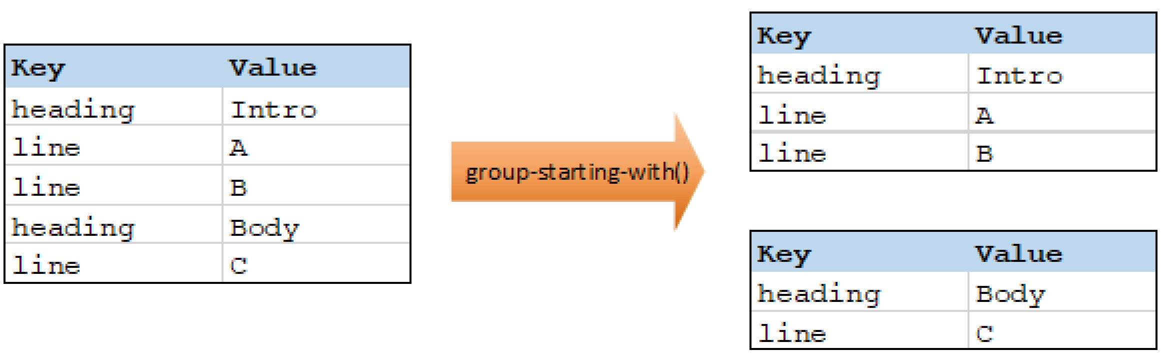
For more information, see the reference to the group-into-blocks[280] function.

group-starting-with

The **`group-starting-with`** function takes a Boolean condition as argument. If the Boolean condition is true, a new group is created, starting with the record that satisfies the condition.



In the example below, the condition is that "Key" must be equal to "heading". This condition is true for the first and fourth records, so two groups are created as a result:

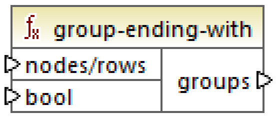


Note: One additional group is created if records exist before the first one that satisfies the condition. For example, if there were more "line" records before the first "heading" record, these would all be placed into a new group.

For more information, see the reference to the group-starting-with[282] function.

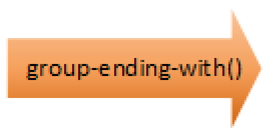
group-ending-with

The **group-ending-with** function takes a Boolean condition as argument. If the Boolean condition is true, a new group is created, ending with the record that satisfies the condition.



In the example below, the condition is that "Key" must be equal to "trailing". This condition is true for the third and fifth records, so two groups are created as a result:

Key	Value
line	A
line	B
trailing	Total 1
line	C
trailing	Total 2



Key	Value
line	A
line	B
trailing	Total 1

Key	Value
line	C
trailing	Total 2

Note: One additional group is created if records exist after the last one that satisfies the condition. For example, if there were more "line" records after the last "trailing" record, these would all be placed into a new group.

For more information, see the reference to the group-ending-with 279 function.

5.7.1 Example: Grouping Records by Key

This example shows you how to group records with the help of the **group-by** function, and also illustrates how to aggregate data. This example is accompanied by a demo mapping available at the following path: **<Documents>\Altova\MapForce2023\MapForceExamples\GroupTemperaturesByYear.mfd**. This mapping reads data from an XML file that contains a log of monthly temperatures, as illustrated in the code listing below:

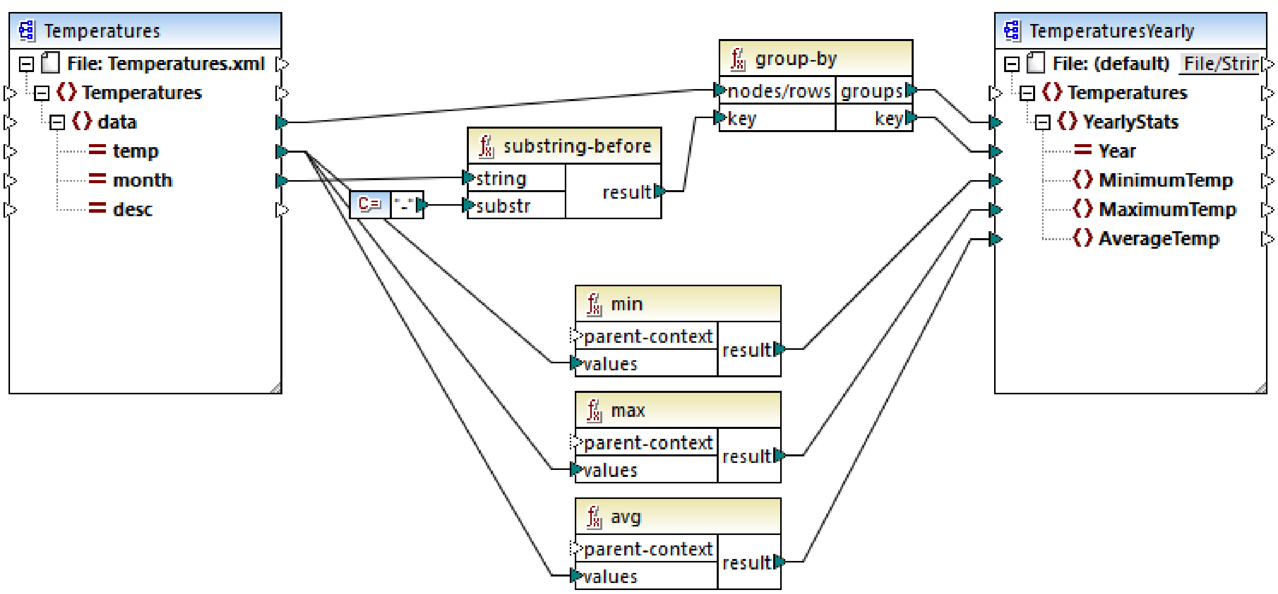
```
<Temperatures>
   <data temp="-3.6" month="2006-01" />
   <data temp="-0.7" month="2006-02" />
   <data temp="7.5" month="2006-03" />
   <data temp="12.4" month="2006-04" />
   <data temp="16.2" month="2006-05" />
   <data temp="19" month="2006-06" />
   <data temp="22.7" month="2006-07" />
   <data temp="23.2" month="2006-08" />
   <data temp="18.7" month="2006-09" />
```

```
    <data temp="11.2" month="2006-10" />
    <data temp="9.1" month="2006-11" />
    <data temp="0.8" month="2006-12" />
    <data temp="-3.2" month="2007-01" />
    <data temp="-0.3" month="2007-02" />
    <data temp="6.5" month="2007-03" />
    <data temp="10.6" month="2007-04" />
    <data temp="19" month="2007-05" />
    <data temp="20.3" month="2007-06" />
    <data temp="22.3" month="2007-07" />
    <data temp="20.7" month="2007-08" />
    <data temp="19.2" month="2007-09" />
    <data temp="12.9" month="2007-10" />
    <data temp="8.1" month="2007-11" />
    <data temp="1.9" month="2007-12" />
</Temperatures>
```

The business requirement of this mapping is two-fold:

1. Group temperatures of each year together.
2. Find out the minimum, maximum, and the average temperature of each year.

To achieve the first goal, the mapping calls the **group-by** 277 function. To achieve the second goal, it calls the **min** 231, **max** 230, and **avg** 228 aggregation functions. All of these functions are MapForce built-in functions, and you can add them to any mapping by dragging them from the Libraries window, see How to Add a Function to the Mapping 191.



GroupTemperaturesByYear.mfd

The way MapForce executes a mapping (and the recommended approach to start reading one) is by looking at the topmost item of the target component. In this example, an **YearlyStats** item will be created for each group returned by the **group-by** function. The **group-by** function takes as first argument all **data** items from the source and groups them by whatever is connected to the **key** input. Since the requirement is to group temperatures by year, the year must be obtained first. To achieve this, the **substring-before** [302] function extracts the year part from the **month** attribute of each **data** element. Namely, it takes as argument the value of **month** and returns the part before the first occurrence of **substr**. As illustrated above, in this example, **substr** is set to the dash character; therefore, if given the value "2006-01", the function will return "2006".

Finally, the values of **MinimumTemp**, **MaximumTemp**, and **AverageTemp** are obtained by connecting these items with the respective aggregate functions: **min**, **max**, and **avg**. All three functions take as input the sequence of temperatures read from the source component. These functions do not need a **parent-context** argument, because they already work in the context of each group. In other words, there is a parent connection —from **data** to **YearlyStats**— which provides the context for each aggregation function to work on.

To preview the mapping output, click the **Output** tab. Notice that the number of groups coincides with the number of years obtained by reading the source file, for example:

```
<Temperatures>
   <YearlyStats Year="2006">
      <MinimumTemp>-3.6</MinimumTemp>
      <MaximumTemp>23.2</MaximumTemp>
      <AverageTemp>11.375</AverageTemp>
   </YearlyStats>
   <YearlyStats Year="2007">
      <MinimumTemp>-3.2</MinimumTemp>
      <MaximumTemp>22.3</MaximumTemp>
      <AverageTemp>11.5</AverageTemp>
   </YearlyStats>
</Temperatures>
```

Note: For simplicity, the code listings above contain less data than the actual input and output used by the demo mapping.

6 Functions

In MapForce, you can use the following categories of functions to transform data according to your needs:

- **MapForce built-in functions** — these functions are predefined in MapForce and you can use them in your mappings to perform a wide range of processing tasks that involve strings, numbers, dates, and other types of data. You can also use them to perform grouping, aggregation, auto-numbering, and various other tasks. For reference to all available built-in functions, see Function Library Reference 225.
- **User-defined functions (UDFs)** — these are MapForce functions that you can create yourself, using as basis the native component kinds and built-in functions already available in MapForce, see User-Defined Functions 198.
- **Custom functions** — these are functions that you can import from external sources such as XSLT libraries and adapt to MapForce. Note that, in order to be reusable in MapForce, your custom functions must return data of simple type (such as string or integer) and they must also take parameters of simple type. For more information, see Importing Custom XSLT Functions 213

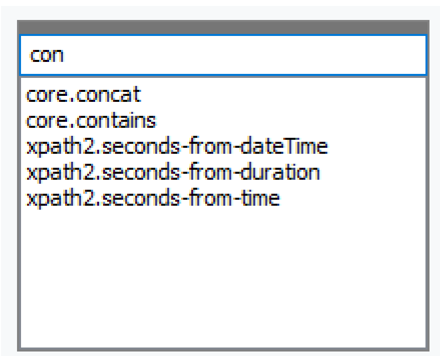
6.1 Functions Basics

The subsections below give an overview of basic function-related actions. The functions you see in the **Libraries** window depend on the transformation language you have selected. For more information, see Transformation Languages 16.

Add function

MapForce includes a large number of built-in functions that you can add to a mapping. For more information about all available built-in functions, see Function Library Reference 225. To add a function to a mapping, you can choose one of the following methods:

- Press and hold 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 at the bottom of the window.
- Double-click anywhere in the empty area of the mapping and start typing the function name (*see screenshot below*). To see a tooltip with more details about a function, click this function in the list. To add a function to your mapping, double-click the relevant function in the combo box.



Add a UDF

You can also add user-defined functions (UDFs) to your mapping using the same approaches as described above if (i) a UDF has already been created in the same mapping or (ii) you have imported a mapping that contains a UDF as a local or global library.

Add a constant

Constants enable you to supply custom text and numbers to a mapping. To add a constant to your mapping, you can choose one of the following options:

- Right-click anywhere in the empty mapping area and select **Insert Constant** from the context menu. Enter the value and select one of the data types: *String*, *Number*, or *All other*.
- Select the **Insert | Constant** menu command. Enter the value and select one of the data types: *String*, *Number*, or *All other*.
- Click the **Constant** toolbar command. Enter the value and select one of the data types: *String*, *Number*, or *All other*.
- Double-click anywhere in the empty mapping area. Type the double quotation mark followed by the constant value. The closing double quotation mark is optional. To add a numeric constant, just type the number.

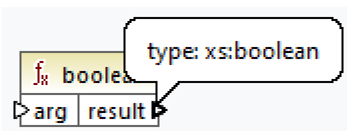
Search for a function

To search for a function in the **Libraries** window, start typing the function name in the text field at the bottom of the 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 *Search in function descriptions* option. To cancel the search, press the **Esc** key or click ✕.

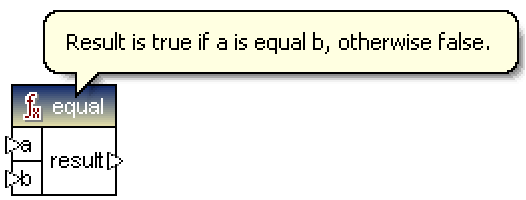
To find all occurrences of a function within the currently active mapping, right-click the function name in the **Libraries** window and select **Find All Calls** from the context menu. The search results are displayed in the **Messages** window.

View a function's type and description

To view the data type of a function's input/output argument, move your mouse over the argument part of a function (*see screenshot below*). Make sure that the INFO (**Show tips**) toolbar button is enabled.

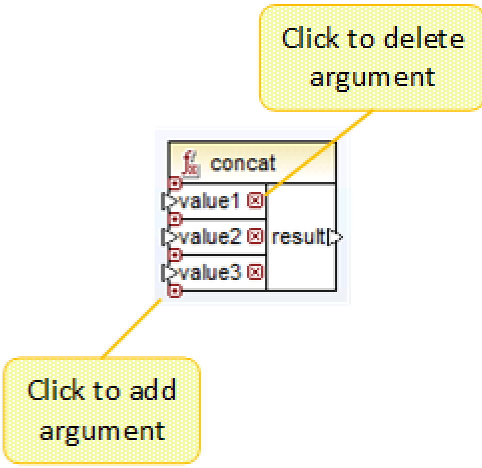


To view the description of a function, move the mouse over the function's header (*see screenshot below*). Make sure that the INFO (**Show tips**) toolbar button is enabled.



Add/delete function arguments

For some MapForce built-in functions, it is possible to add as many parameters as you need for your mapping purposes. One such example is the **concat** 296 function. To add or delete function arguments (for functions that support that), click **Add parameter** (⊕) or **Delete parameter** (⊠) next to the parameter you want to add or delete, respectively (*see below*). Moving a connection to the ⊕ symbol adds another parameter and connects this parameter.


Click to delete argument
concat
value1
value2
value3
result
Click to add argument

6.2 Manage Function Libraries

In MapForce, you can import and use the following kinds of libraries in a mapping:

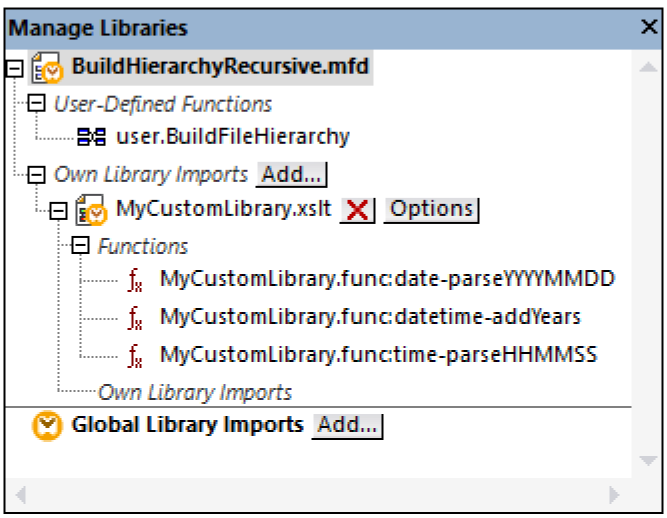
- Any mapping design files (*.mfd) that contain user-defined functions (UDFs). This specifically refers to mapping files that contain UDFs created with MapForce, using the MapForce built-in functions and components as building blocks. For further information, see Creating User-Defined Functions 199.
- Custom XSLT files that contain functions. This refers to XSLT functions written outside of MapForce that qualify for import into MapForce as described in Importing Custom XSLT Functions 213.

Manage Libraries window

You can view and manage all libraries used by a mapping file from the Manage Libraries window. This includes UDFs and custom libraries.

By default, the **Manage Libraries** window is not visible. To display it, do one of the following:

- In the **View** menu, click **Manage Libraries**.
- Click **Add/Remove Libraries** at the bottom of the **Libraries** window.



You can choose to view UDFs and libraries only for the mapping document that is currently active or for all open mapping documents. To view imported functions and libraries for all of the currently open mapping documents, right-click inside the window and select **Show Open Documents** from the context menu.

To display the path of the open mapping document instead of the name, right-click inside the window and select **Show File Paths** from the context menu.

Data displayed in the Manage Libraries window is organized as a tree hierarchy as follows:

- Any currently open mapping documents are displayed as top-level entries. Each entry has two branches: **User-Defined Functions** and **Own Library Imports**.
  - The **User-Defined Functions** branch displays any UDFs contained in that document.
  - The **Own Library Imports** branch displays libraries imported *locally* into the current mapping document. The term "libraries" means other mapping documents (.mfd files that contain user-

defined functions) or custom external libraries written in XSLT 1.0, XSLT 2.0, XQuery 1.0*, Java*, C#*, or .mff files mentioned previously. Note that the **Own Library Imports** structure could be several levels deep, since any mapping document may import any other mapping document as a library.

- The **Global Library Imports** entry encloses any custom libraries that you have imported *globally* at application level. Again, in case of .mfd files, the structure could be several levels deep, for the reasons mentioned above.

** These languages are supported only in MapForce Professional or Enterprise edition.*

Note: The XSLT, XQuery, C#, and Java libraries may have dependencies of their own. Such dependencies are not displayed in the Libraries window.

Context menu commands

You can quickly perform various operations against objects in the Manage Libraries window by right-clicking an object and selecting one of the following context menu options:

Command	Description	Applicable for
Open	Opens the mapping.	Mappings
Add	Opens a dialog box where you can browse for a custom library of functions.	Own Library Imports
Locate Function in Libraries Window	Changes focus to the Libraries window, and selects the function.	Functions
Cut, Copy, Delete	These standard Windows commands are applicable only to MapForce user-defined functions. You cannot copy-paste functions from external XSLT files or other library kinds.	User-defined functions
Paste	Lets you paste a user-defined function that was previously copied to clipboard into the current library.	Libraries (UDF)
Options	Opens a dialog box where you can set or change options for the current library.	Libraries
Show All Open Documents	When this option is switched on, the Manage Libraries window will display all currently open mappings. This is typically useful if you need to copy-paste functions between mappings. Otherwise, only the mapping that is currently in focus is shown.	Always
Show File Paths	When this option is switched on, objects in the Manage Libraries window are displayed with their full path. Otherwise, only the object name is shown.	Always

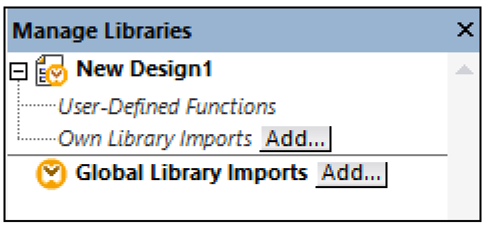
6.2.1 Local and Global Libraries

You can import libraries *locally* or *globally*. Global imports are at application level. If a library was imported globally, you can use its functions from any mapping.

Local imports are at mapping file level. For example, let's suppose that, while working on mapping **A.mfd**, you decide to import all user-defined functions from mapping **B.mfd**. In this case, mapping **B.mfd** is considered to be imported as a local library into **A.mfd** and you can use functions from **B.mfd** in **A.mfd** as well. Likewise, if you import functions from an XSLT file into **A.mfd**, this is also a local import.

You can view and manage all local and global imports from the Manage Libraries window. To import a library, do one of the following:

1. Click the **Add/Remove Libraries** button at the bottom of the Libraries window[22]. The **Manage Libraries** window opens (*see screenshot below*).



2. To import functions as a *local* library (in the scope of the current document only), click **Add** under the current mapping name. To import functions as a *global* library (at program level), click **Add** next to **Global Library Imports**. When you import a library *locally*, you can set the path of the library file to be relative to the mapping file. With globally imported libraries, the path of the imported library is always absolute.

Conflicting function names

You may come across situations where the same function name is defined at any of the following levels:

- in the main mapping
- in a library that was imported locally
- in a library that was imported globally

When it encounters such cases, MapForce will attempt to call the function exactly in the order above, to prevent ambiguity. That is, the function defined directly in the mapping takes precedence if the same function name exists in a locally imported library. Also, the function imported locally takes precedence over the function imported globally (assuming that both functions have the same name).

If multiple functions with the same name exist, only the "winning" function will be called, according to the rule above; any other ambiguous function names will be blocked. Such blocked functions appear as grayed out in the Libraries window, and it is not be possible to use them in the mapping.

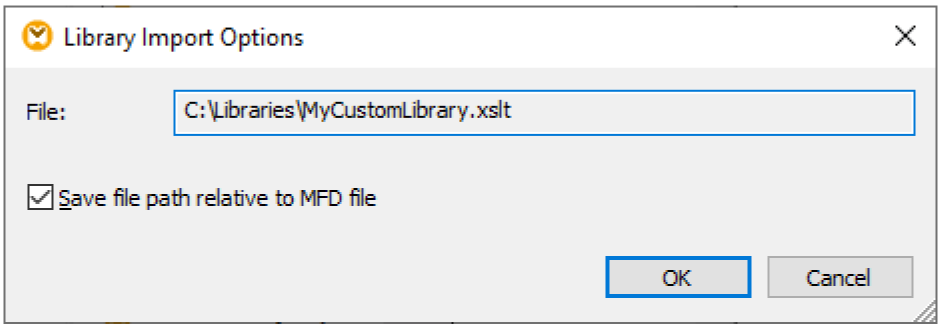
6.2.2 Relative Library Paths

You can set the path of any imported library file to be relative to the mapping design file (.mfd), provided that the library was imported locally (not globally), as described in Local and Global Libraries[196].

Setting a relative library path is applicable only for those libraries that were imported *locally* at document level. If a mapping was imported *globally* at program level, its path is always absolute.

To set a library path as relative to the mapping design file:

1. Click **Add/Remove Libraries** at the base of the Libraries window. The Manage Libraries window [24] opens.
2. Click **Options** next to the library of interest. (Alternatively, right-click the library, and select **Options** from the context menu.)



3. Select the **Save file path as relative to MFD file** check box.

Note: If the check box is grayed out, make sure that the library was indeed imported locally, and not globally.

When the check box is selected, MapForce will keep track and update the path to any referenced library files when you save the mapping file to a new directory using the **Save as** menu command. Also, if the library files are in the same directory as the mapping file, the path reference will not be broken when you move the entire directory to a new location on the disk, see also Using Relative Paths on a Component [73].

Note that the **Save file path as relative to MFD file** check box specifies that paths are *relative to the mapping file*, and it does not affect paths in generated code. For information about how library references are handled in generated code, see Paths in Various Execution Environments [76].

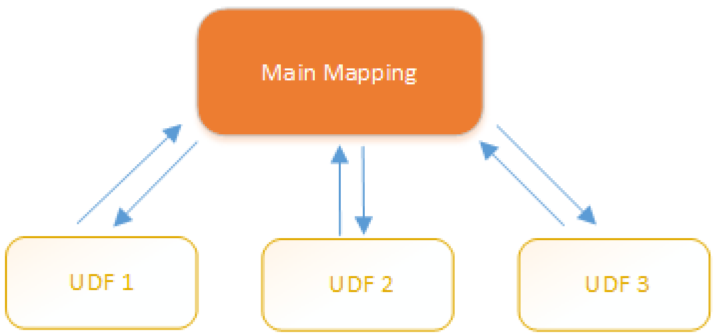
6.3 User-Defined Functions

User-Defined Functions (UDFs for short) are custom functions that are defined once and can be reused multiple times within the same mapping or across multiple mappings. UDFs are like mini-mappings themselves: They typically consist of one or more input parameters, some intermediary components to process data, and an output to return data to the caller. The caller is the main mapping or another UDF.

Advantages of UDFs

UDFs have the following advantages:

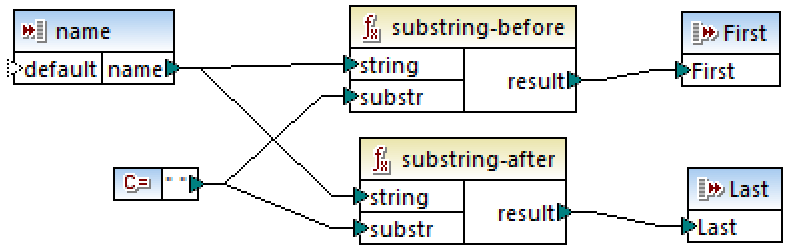
- UDFs are reusable within one mapping or across multiple mappings.
- UDFs can make your mapping easier to read: For example, you can package parts of the mapping into smaller components and abstract away the implementation details. The diagram below illustrates this principle.



- UDFs are flexible functions that enable you to process strings, numbers, dates, and other data in a custom way that extends the built-in MapForce functions. For example, you might want to concatenate or split text in a particular way, perform some advanced calculations, manipulate dates and times, and so on.
- Another common use of UDFs is to look up a field in an XML file, database or some other data format supported by your MapForce edition and present this data in a convenient way. For details, see Look-up Implementation 210.
- UDFs can be called recursively (i.e., the UDF calls itself). This requires that the UDF be defined as a regular (not inline) function 201. Recursive UDFs 207 can fulfill various advanced mapping requirements, such as iterating over data structures having a depth of *N* children, where *N* is not known in advance.

Example

Below is an example of a simple UDF that splits a string into two separate strings. This UDF is part of a larger mapping called **`MapForceExamples\ContactsFromPO.mfd`**. The UDF takes the name as a parameter (e.g., `Helen Smith`), applies the built-in functions **`substring-before`** and **`substring-after`**, and then returns two values: `Helen` and `Smith`.



In this section

This section explains how to work with UDFs and is organized into the following topics:

- UDF Basics 199
- UDF Parameters 204
- Recursive UDFs 207
- Look-up Implementation 210

6.3.1 UDF Basics

This topic explains how to create, import, edit, copy-paste, and delete user-defined functions (UDFs for short).

Create a UDF

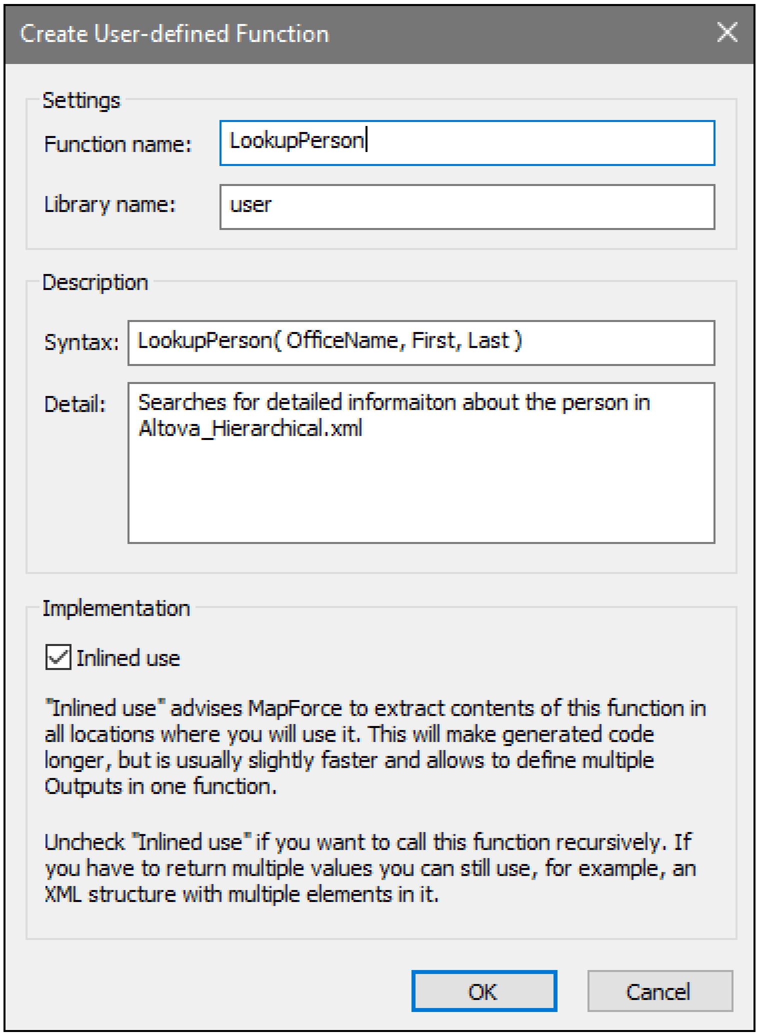
This subsection explains how to create a UDF from scratch and from already existing components. The minimum requirement is one output component to which some data is connected. For input parameters, a function can have zero, one or more inputs. The input and output parameters can be of simple type (e.g., a string) or complex type (a structure). For more information about simple and complex parameters, see UDF Parameters 204.

UDF from scratch

To create a UDF from scratch, follow the instructions below:

1. Select **Function | Create User-Defined Function**. Alternatively, click the toolbar button.

2. Enter the relevant information into the **Create User-defined Function** dialog (*see screenshot below*).



The available options are listed below.

- *Function Name:* Mandatory field. For the name of your UDF, you can use the following characters: alphanumeric characters (a-z, A-Z, 0-9), an underscore (_), a hyphen/dash (-), and a colon (:).
- *Library Name:* Mandatory field. This is the name of a function library (in the Libraries window[22]) in which your function will be saved. If you do not specify the name of a library, the function will be placed into a default library called `user`.
- *Syntax:* Optional field. Enter some text that concisely describes the syntax of the function (e.g., expected parameters). This text will be displayed next to the function in the **Libraries** window and does not affect the implementation of the function.
- *Detail:* Optional field. This description will be displayed when you move the cursor over the function in the **Libraries** window or in other contexts.
- *Inlined use:* Select this check box if the function should be created as inline. For more information, see *Regular vs. Inline UDFs* below.

3. Click **OK**. The function becomes immediately visible in the **Libraries** window under the library name specified above. The mapping window is now redrawn to allow you to create a new function (this is a standalone mapping referred to as *the function's mapping*). The function's mapping includes an output component by default.

4. Add all the required components to the function's mapping. You can do this in the same way as for a standard mapping.

To use the UDF in a mapping, drag the UDF from the **Libraries** window onto the main mapping area. See also *Call and import UDFs* below.

UDF from existing components
To create a UDF from existing components, take the steps below:

1. Select the relevant components on the mapping by making a rectangular selection with the mouse. You can also select multiple components by clicking each one while holding the **Ctrl** key pressed.
2. Select the menu command **Function | Create User-Defined Function from Selection**. Alternatively, click the toolbar button.
3. Follow Steps 2-4 from *UDF From Scratch*.

Regular vs. Inline UDFs
There are two types of UDFs: *regular* and *inline*. You can specify the type of your UDF when you create it. Inline and regular functions behave differently in terms of code generation, recursiveness, and the ability to have multiple output parameters. The table below summarizes the main differences between regular and inline UDFs.

Inline functions (dashed border)	Regular functions (solid border)
With inline functions, the UDF code is inserted at all locations where the function is called. If the UDF is called several times, the generated program code would be significantly longer.	The code for the UDF is generated once, and inputs to it are passed as parameter values. If the UDF is called several times, the UDF is evaluated each time with the corresponding parameter values.
Inline functions can have multiple outputs and thus return multiple values.	Regular functions can have only one output. To return multiple values, you can declare the output to be of complex type (e.g., an XML structure), which would allow you to pass multiple values to the caller.
Inline functions cannot be called recursively.	Regular functions can be called recursively.
Inline functions do not support setting a priority context[408] on a parameter.	Regular functions support setting a priority context on a parameter.

Note: Switching a UDF from inline to regular, and vice versa, may affect the mapping context[399], and this may cause the mapping to produce a different result.

Call and import UDFs

After you have created a UDF, you can call it from the same mapping where you created it or from any other mapping.

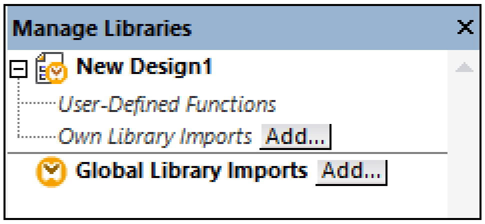
Call UDF from the same mapping
To call a UDF from the same mapping, take steps below:

1. Find the relevant function in the **Libraries** window under the library that you specified when you created the function. To do that, start typing the name in the **Libraries** window.

2. Drag the function from the **Libraries** window into the mapping. You can now connect all the required parameters to the function.

Import UDF from a different mapping
To import a UDF from another mapping, follow the instructions below:

1. Click the **Add/Remove Libraries** button at the bottom of the Libraries window [22]. The **Manage Libraries** window opens (*see screenshot below*).



2. To import functions as a *local* library (in the scope of the current document only), click **Add** under the current mapping name. To import functions as a *global* library (at program level), click **Add** next to **Global Library Imports**. When you import a library *locally*, you can set the path of the library file to be relative to the mapping file. With globally imported libraries, the path of the imported library is always absolute.

3. Browse for the **.mfd** file that contains the UDF and click **Open**. A message box will inform you that a new library has been added and can be accessed in the **Libraries** window.

You can now use any of the imported functions in the current mapping by dragging them from the **Libraries** window into the mapping. For more information about viewing and organizing function libraries, see Manage Function Libraries [194].

Mapping with credentials (Enterprise Edition)
If the imported **.mfd** file contains credentials, these are shown as imported (with a yellow background) in Credentials Manager. By default, imported credentials are not saved with the mapping, but you can optionally create a local copy and save them in the mapping.

Edit UDFs

To edit a UDF, take the steps below:

1. Open the mapping that contains a UDF.
2. Double-click the title bar of the UDF in the mapping to see the function's contents where you can add, edit, or remove components as required.
3. To change the function's properties (e.g., the name or description), right-click an empty area in the mapping and select **Function Settings** from the context menu. Alternatively, click the toolbar button.

You can also edit a function by double-clicking its name in the **Libraries** window. However, only functions in the currently active document can be opened this way. Double-clicking a UDF that was created in another mapping opens that mapping in a new window. If you edit or delete a UDF that was imported into multiple mappings, all these mappings will be affected by the change.

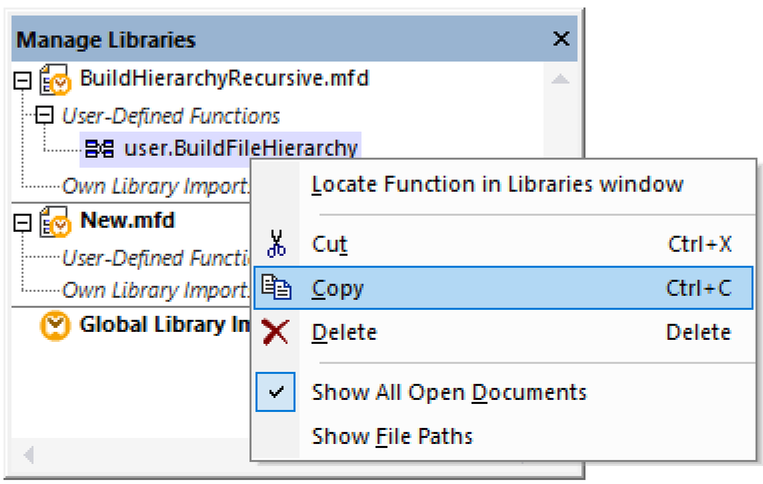
To go back to the main mapping, click the button in the top-left corner of the mapping window.

MapForce allows you to navigate through different mappings and UDFs by using the and toolbar buttons. The corresponding keyboard shortcuts for these buttons are **Alt+Left** and **Alt+Right**, respectively.

Copy-paste UDFs

To copy a UDF and paste it into another mapping, follow the steps below:

1. Open the Manage Libraries window[194].
2. Right-click inside an empty area in the **Libraries** window and select the option **Show All Open Documents**.
3. Open both the source and destination mappings. Make sure that both mappings are saved to disk. This ensures correct resolution of paths. See also Copy-Paste and Relative Paths[74].
4. Right-click the relevant UDF from the source mapping in the **Manage Libraries** window and select **Copy** from the context menu (*see screenshot below*) or press **Ctrl+C**. Leave the **Manage Libraries** window open.



5. Switch to the destination mapping (and the **Manage Libraries** window will change accordingly), right-click *User-Defined Functions*, and select **Paste** from the context menu.

Delete UDFs

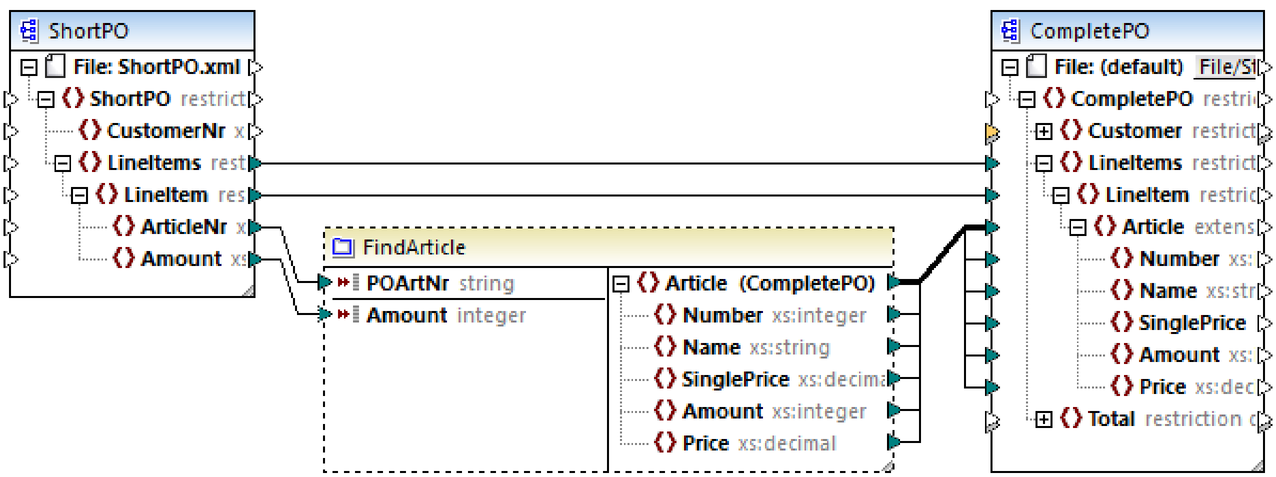
To delete a UDF, take the steps below:

1. Double-click the title bar of the UDF in the mapping.
2. Click the button in the top-right corner of the Mapping window.
3. If the function is used in the currently open mapping, MapForce will ask whether you want to replace all instances with internal components. Click **Yes** if you want to delete the function and replace all instances where it is called with the function's components. This lets you keep the main mapping valid even if the function is deleted. However, if the deleted function is used in any other external mappings, those will not be valid. Click **No** if you want to delete the function and all its internal components permanently. In this case, all the mappings where the function is used will not be valid.

6.3.2 UDF Parameters

When you create a UDF, you must specify what input parameters it should take (if any) and what output it should return. While input parameters are sometimes not necessary, an output parameter is mandatory in all cases. Function parameters can be of simple type (e.g., `string` or `integer`) or a complex structure [204]. For example, the `FindArticle` UDF illustrated below has two input and one output parameters:

- `POArtNr` is an input parameter of type `string`.
- `Amount` is an input parameter of type `integer`.
- `CompletePO` is an output parameter that has a complex XML structure.



Parameter order
When a UDF has multiple input or output parameters, you can change the order in which parameters will appear to the callers of this function. The order of parameters in the function's mapping (starting from the top) dictates the order in which they appear to the callers of this function.

Important

- Input and output parameters are sorted by their position from top to bottom. Therefore, if you move the `input3` parameter to the top in the function's mapping, it will become the first parameter of this function.
- 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.

Complex-type structures

The structures on which a parameter in a UDF can be based are summarized in the list below.

MapForce Basic Edition
- XML Schema Structure

MapForce Professional Edition
- XML Schema Structure
- Database Structure

MapForce Enterprise Edition

- XML Schema Structure
- Database Structure
- EDI Structure
- FlexText Structure
- JSON Schema Structure

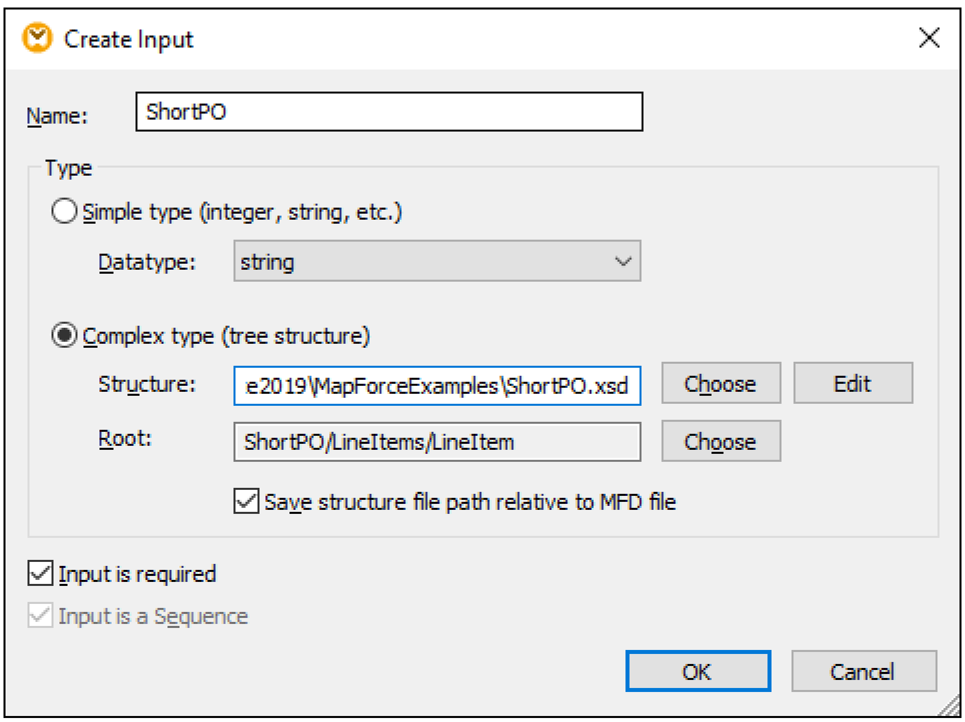
UDFs based on database structures (Professional and Enterprise editions)

MapForce allows you to create DB-based UDF parameters with a tree of related tables. The tree of related tables represents an in-memory structure that has no connection to the database at runtime. This also means that there is no automatic handling of foreign keys and no table actions in parameters or variables.

Add Parameters

To add an input or output parameter, take the following steps:

1. Create a UDF [199] or open an existing one [201].
2. Run the menu command **Function | Insert Input** or **Function | Insert Output** (*see screenshot below*). Alternatively, click (**Insert Input**) or (**Insert Output**) in the toolbar.



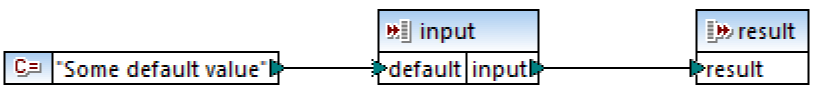
3. Choose whether input or output parameters should be of simple or complex type (*see dialog box above*). See the list of available complex structures above. For example, to create a parameter that is a complex XML type, click **Choose** next to *Structure* and browse for the XML schema that describes the required structure.

If the function's mapping already includes XML schemas, they become available for selection as structures. Otherwise, you can select a new schema that will provide the structure of the parameter. The same is true for databases and other complex structures if they are supported by your MapForce edition. With XML structures, it is possible to select the root element for your structure if the XML schema allows it. To specify the root element, click **Choose** next to *Root* and select the root item from the dialog box that opens.

If selected, the check box *Save structure file path relative to MFD file* will change the absolute path of the structure into a path relative to the current mapping, when you save the mapping. For more information, see Relative and Absolute Paths 73. The check boxes *Input is required* and *Input is a Sequence* are explained below.

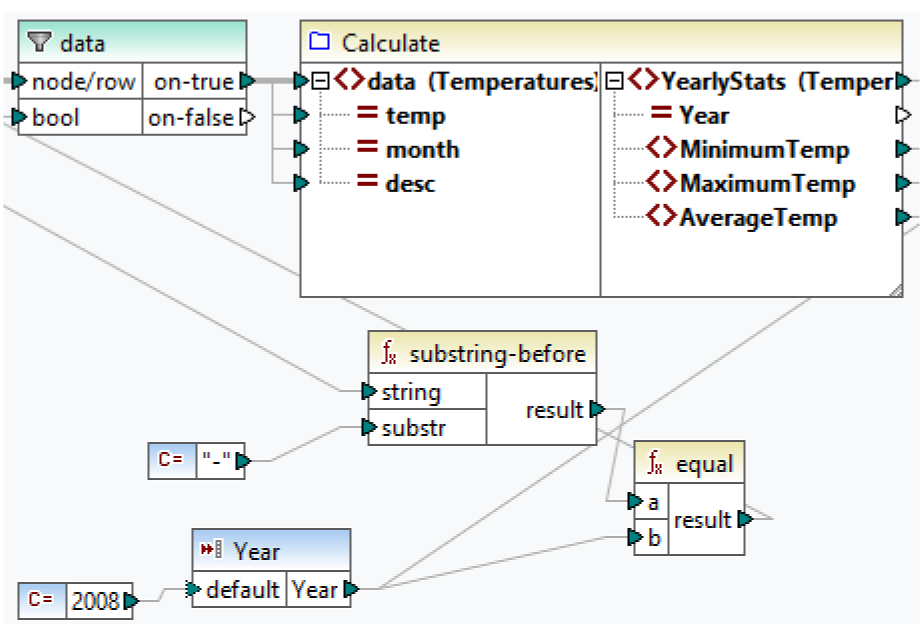
Input is required
To make a parameter mandatory in a UDF, select the *Input is required* check box (*see dialog box above*). If you clear the *Input is required* check box, the parameter will become optional and have a dashed border in the mapping.

You can also specify a default parameter value by connecting it to the `default` input of a parameter (*see example below*). The default value will apply only if there is no other value. If the optional parameter receives a value when the function is called, that value takes precedence over the default.

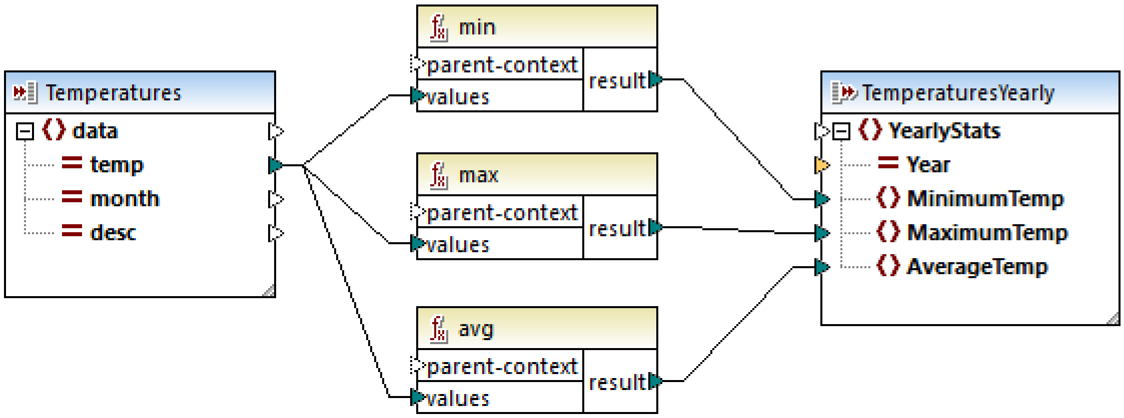


Input is a sequence
You can optionally define whether a function's parameter should be treated as a single value (default option) or as a sequence. A sequence is a range of zero or more values. A sequence might be useful when your UDF expects input data as a sequence in order to calculate values in that sequence, for example, by calling functions such as **avg**, **min**, **max**. To treat the input of the parameter as a sequence, select the *Input is sequence* check box. Note that this check box is enabled only if the UDF is regular 201.

The usage of a sequence is illustrated in the following mapping: **MapForceExamples\InputIsSequence.mfd**. In the extract of this mapping (*see screenshot below*), the `data` filter is connected to the UDF called `Calculate`. The filter's output is a sequence of items. Therefore, the input parameter of the function is set to be a sequence.



Illustrated below is the implementation of the `Calculate` function that aggregates all the sequence values: It runs the **`avg`**, **`min`**, and **`max`** functions on the input sequence. To see the internal structure of the `Calculate` function, double-click the header of the `Calculate` component in the mapping above.



As a rule of thumb, the input data (sequence or non-sequence) determines how often the function is called:

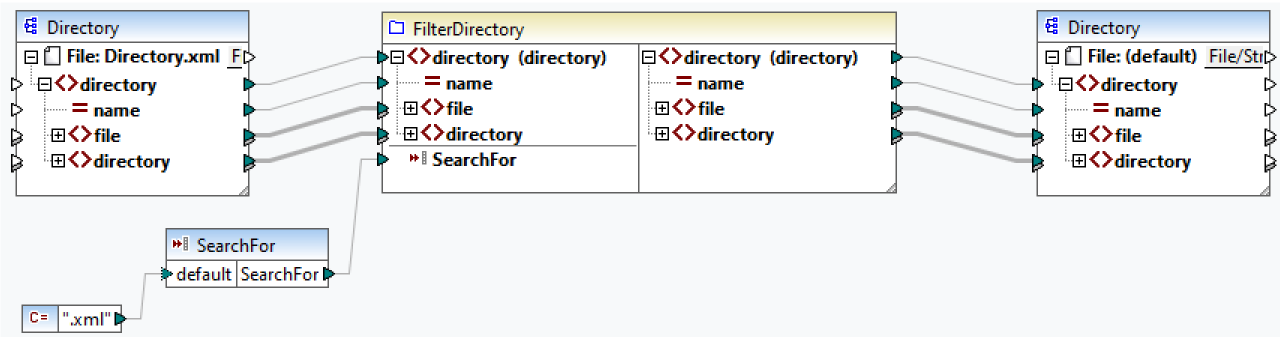
- When input data is connected to a *sequence* parameter, the UDF is called only *once*, and the complete sequence is passed into the UDF.
- When input data is connected to a *non-sequence* parameter, the UDF is called *once for each single item* in the sequence.
- If you connect an empty sequence to a non-sequence parameter, 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, use the `substitute-missing` 295 function before the function input to ensure that the sequence is never empty. Alternatively, set the parameter to a sequence and add handling for the empty sequence inside the function.

The *Output is a Sequence* check box may be required for output parameters, too. When a function passes a sequence of multiple values to its output component, and the output component is not set to sequence, the function will return only the first item in the sequence.

6.3.3 Recursive UDFs

This topic explains how to search for data in a source XML file with the help of a recursive UDF. To test the recursive UDF, you will need the following mapping: **`MapForceExamples\RecursiveDirectoryFilter.mfd`**. In the mapping below, the `FilterDirectory` UDF receives data from the source file `Directory.xml` and the simple input component `SearchFor` that supplies the `.xml` extension. After the data has been processed by the UDF, it is mapped to the target file.

The main mapping (*see screenshot below*) describes the general mapping layout. The way the UDF processes the data is defined separately, in the function's mapping (*See UDF Implementation below*).



Goal

Our goal is to list files with a **`.xml`** extension in the output while preserving the entire directory structure. The subsections below explain the mapping process in detail.

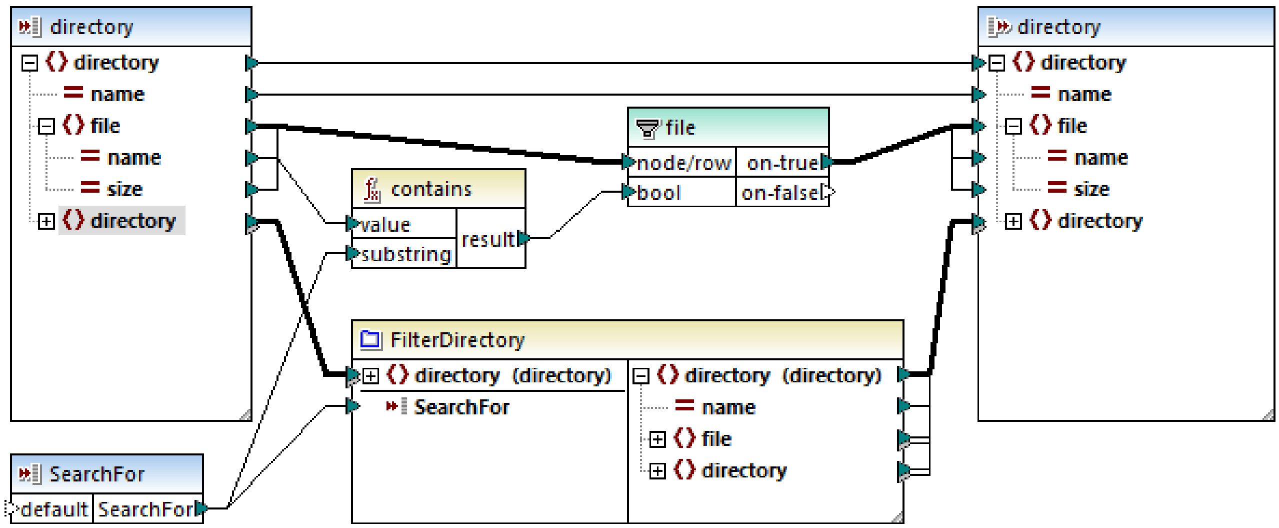
Source file

Below is an extract from the source XML file (**`Directory.xml`**) that contains information about files and directories. Note that the files in this listing have different extensions (e.g., **`.xml`**, **`.dtd`**, **`.sps`**). According to the schema (**`Directory.xsd`**), the `directory` element can have `file` children and `directory` children. All `directory` elements are recursive.

```
<directory name="ExampleSite">
	<file name="blocks.sps" size="7473"/>
	<file name="blocks.xml" size="670"/>
	<file name="block_file.xml" size="992"/>
	<file name="block_schema.xml" size="1170"/>
	<file name="contact.xml" size="453"/>
	<file name="dictionaries.xml" size="206"/>
	<file name="examplesite.dtd" size="230"/>
	<file name="examplesite.spp" size="1270"/>
	<file name="examplesite.sps" size="20968"/>
	...
	<directory name="output">
		<file name="examplesite1.css" size="3174"/>
		<directory name="images">
			<file name="blank.gif" size="88"/>
			<file name="block_file.gif" size="13179"/>
			<file name="block_schema.gif" size="9211"/>
			<file name="nav_file.gif" size="60868"/>
			<file name="nav_schema.gif" size="6002"/>
		</directory>
	</directory>
</directory>
```

UDF implementation

To see the internal implementation of the UDF, double-click its header in the main mapping. The UDF is recursive, that is, it includes a call to itself. Because it is connected to the recursive element `directory`, this function will be called as many times as there are nested `directory` elements in the source XML instance. To support recursive calls, the function must be regular 201.



The implementation of the UDF consists of two parts: (i) defining the files and (ii) defining the directory to be searched.

Defining files

The UDF processes the files as follows: The **contains** function checks whether the first string (the file name) contains the substring `.xml` (supplied by the simple input component `SearchFor`). If the function returns `true`, the file name with a `.xml` extension is written to the output.

Processing child directories

Child directories of the current directory are sent as input to the current UDF. The UDF thus iterates through all `directory` elements and check whether files with the **`.xml`** extension exist.

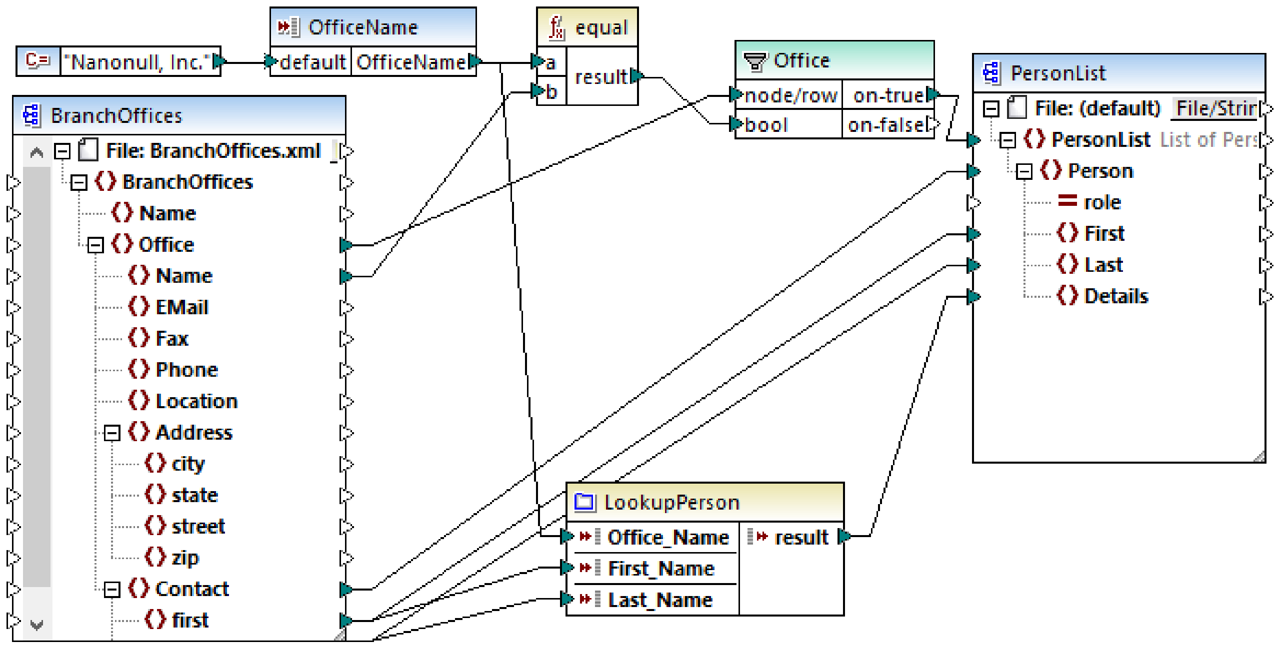
Output

When you click the **Output** pane, MapForce will display only files with the **`.xml`** extension (*see extract below*).

```
<directory name="ExampleSite">
       <file name="blocks.xml" size="670"/>
       <file name="block_file.xml" size="992"/>
       <file name="block_schema.xml" size="1170"/>
       <file name="contact.xml" size="453"/>
       ...
       <directory name="output">
              <directory name="images"/>
       </directory>
</directory>
```

6.3.4 Look-up Implementation

This topic explains how to look up data about employees and present this information in a suitable way. To test the look-up implementation, you will need the following mapping:
MapForceExamples\PersonListByBranchOffice.mfd.



Goals
Our goals are as follows:

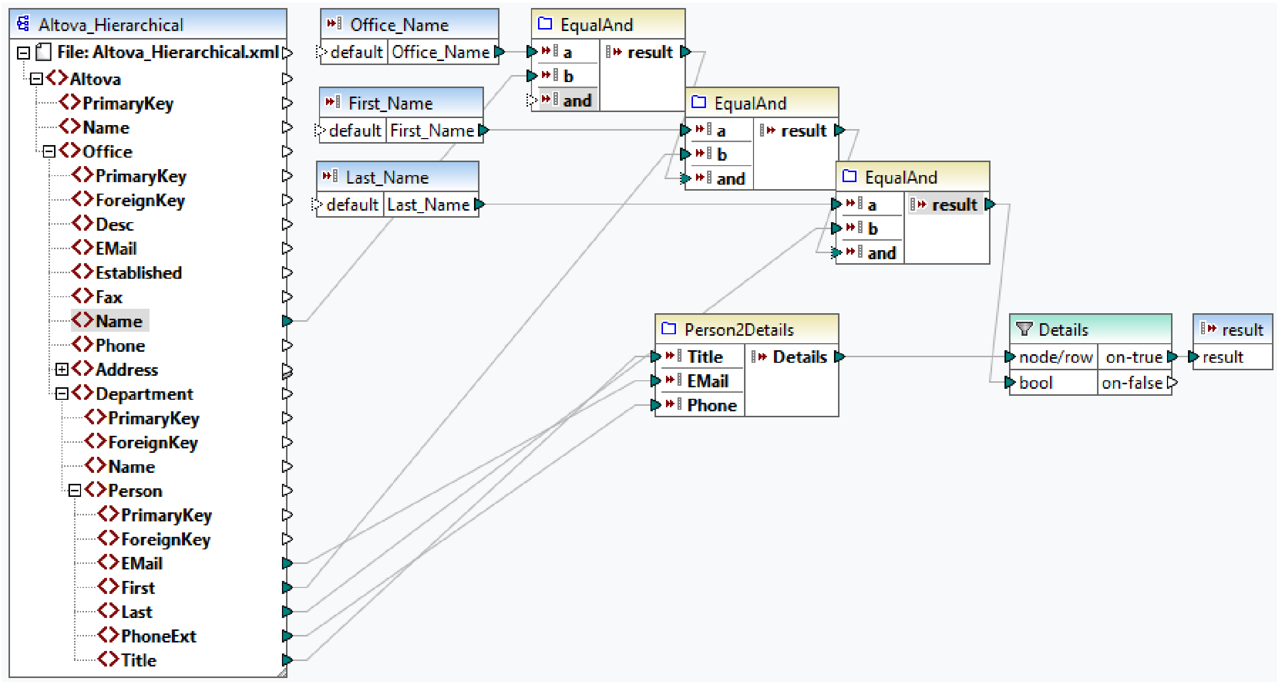
- To look up data about each employee (their phone extension, email address, and title) in a separate XML file.
- To present this data as a comma-separated list and map this list to the `Details` element of the target XML file.
- To extract information about employees only from one branch office called Nanonull, Inc.

To achieve these goals, we have designed our mapping in the following way:

- To filter only employees from Nanonull, Inc., the mapping uses the `Office` filter.
- To look up information about employees in a different XML file, the mapping calls the `LookupPerson` UDF. The implementation of this UDF is described in the subsection below.
- To process employee data, the `LookupPerson` function internally calls other functions that retrieve and concatenate information about each employee. All these operations are in the function's mapping and not visible in the main mapping. The `LookupPerson` function then maps the employee data to the `Details` element in `PersonList`.

LookupPerson implementation

The look-up functionality is provided by the `LookupPerson` function, whose definition is illustrated below. To see the internal implementation of the UDF, double-click its header in the main mapping.

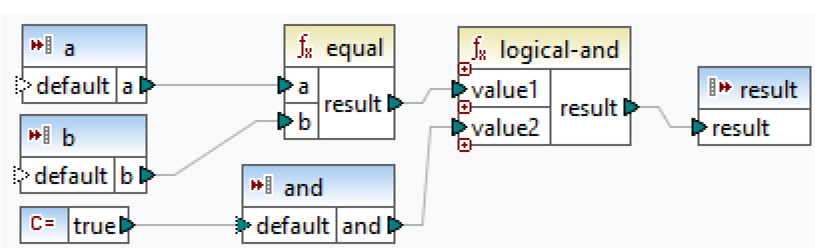


The UDF is defined as follows:

- The data is retrieved from the XML file **`Altova_Hierarchical.xml`**: (i) the name of the office and first and last names of employees, which are used to select employees only from Nanonull, Inc., and (ii) the email, title, and phone extension that are concatenated into one string. The definitions of the `EqualAnd` and `Person2Detail` functions are described below.
- The UDF also has three input parameters that provide the look-up values `Office_Name`, `First_Name`, and `Last_Name`. The value of the `Office_Name` parameter is retrieved from the `OfficeName` input from the main mapping, and the values of `First_Name` and `Last_Name` are supplied by the `BranchOffices` component from the main mapping.
- The value of the `EqaulAnd` function (`true` or `false`) is passed to the `Details` filter each time a new employee's details (title, email, phone) are processed. When the `Details` filter gets the value `true`, the look-up operation is successful and the employee's details are retrieved and returned to the main mapping. Otherwise, the next item in the context is examined, and this procedure continues until the loop finishes.

EqualAnd implementation

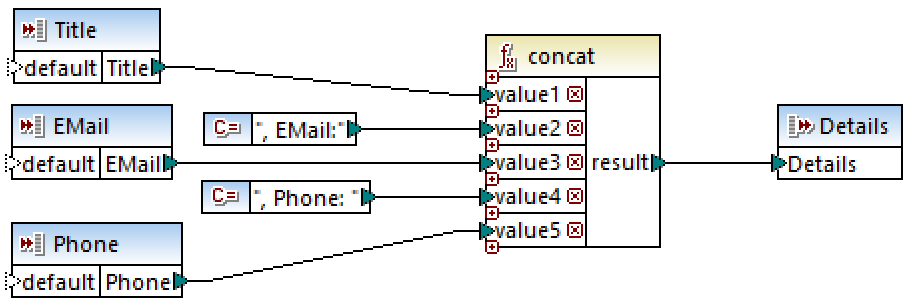
The `EqualAnd` function (*see below*) is a separate UDF defined inside the `LookupPerson` UDF. To see the internal structure of the `EqualAnd` UDF, double-click the function's header.



The `EqualAnd` UDF first checks whether `a` equals `b`; if the result is `true`, it is passed as the first parameter of the **`logical-and`** function. If both values are true in the **`logical-and`** function, the result is also true and is passed on to the next `EqualAnd` function. The result of the third `EqaulAnd` function (*see LookupPerson UDF above*) is passed on to the `Details` filter.

Person2Detail implementation

The `Person2Details` UDF is another function inside the `LookupPerson` UDF. The `Person2Details` UDF (*see below*) concatenates three values (retrieved from **`Altova_Hierarchical.xml`**) and two text constants.



Output

When you click the **Output** pane, MapForce will display first and last names, and details of employees only from Nanonull, Inc (*see extract below*).

```
<PersonList>
       <Person>
              <First>Vernon</First>
              <Last>Callaby</Last>
              <Details>Office Manager, EMail:v.callaby@nanonull.com, Phone:
582</Details>
       </Person>
       <Person>
              <First>Frank</First>
              <Last>Further</Last>
              <Details>Accounts Receivable, EMail:f.further@nanonull.com, Phone:
471</Details>
       </Person>
       ...
</PersonList>
```


6.4 Custom Functions

This section explains how to import custom XSLT[213] functions.

6.4.1 Import Custom XSLT Functions

You can extend the XSLT 1.0, XSLT 2.0, and XSLT 3.0 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. Click the **Add/Remove Libraries** button at the bottom of the Libraries window[22]. The **Manage Libraries** window opens (*see screenshot below*).

2. To import functions as a *local* library (in the scope of the current document only), click **Add** under the current mapping name. To import functions as a *global* library (at program level), click **Add** next to **Global Library Imports**. When you import a library *locally*, you can set the path of the library file to be relative to the mapping file. With globally imported libraries, the path of the imported library is always absolute.

3. Browse for the .xsl file that contains the functions, and click **Open**. A message box appears informing you that a new library has been added.

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 have selected XSLT as a transformation language[16]. See also Managing Function Libraries[194].

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 96.

Data types 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 Functions 214
Example: Summing Node Values 217
XSLT 1.0 engine implementation 482
XSLT 2.0 engine implementation 482

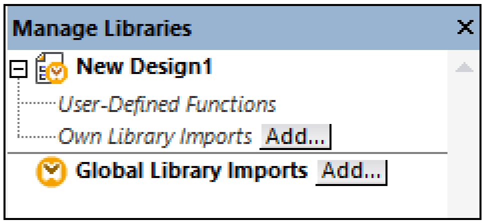
6.4.1.1 Example: Adding Custom XSLT Functions

This example illustrates how to import custom XSLT 1.0 functions into MapForce. The files needed for this example are available in the following folder: **`C:\Users\<username>\Documents\Altova\MapForce2023\MapForceExamples`**.

- `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)

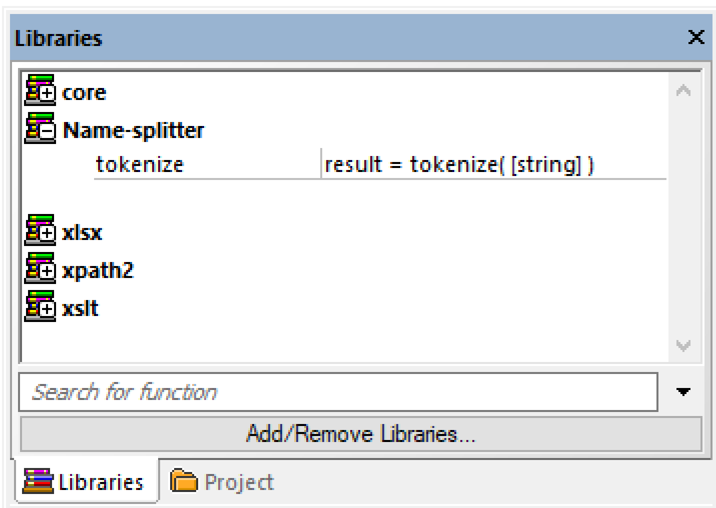
To add a custom XSLT function:

1. Click the **Add/Remove Libraries** button at the bottom of the Libraries window 22. The **Manage Libraries** window opens (*see screenshot below*).



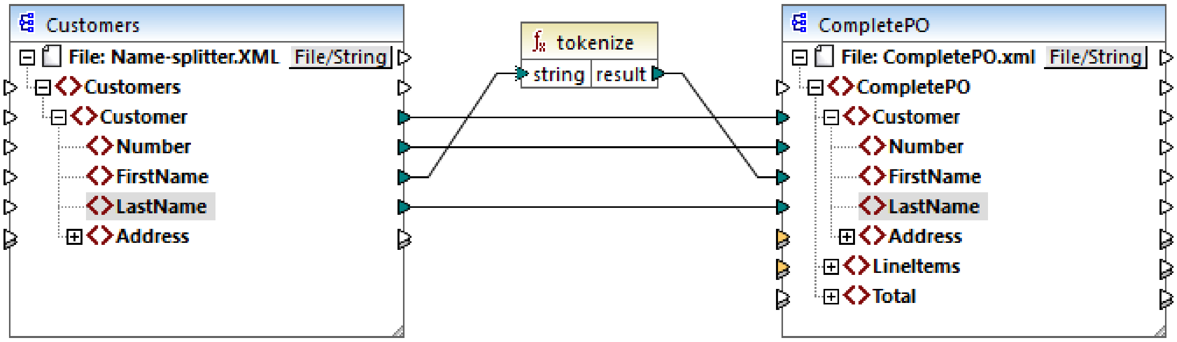
2. To import functions as a *local* library (in the scope of the current document only), click **Add** under the current mapping name. To import functions as a *global* library (at program level), click **Add** next to **Global Library Imports**. When you import a library *locally*, you can set the path of the library file to be relative to the mapping file. With globally imported libraries, the path of the imported library is always absolute.

3. 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`, and click **Open**. A message box appears informing you that a new library has been added, and 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:

1. Drag the **`tokenize`** function into the Mapping window and map the items as show below.



2. Click the **XSLT** tab to see the generated XSLT code.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--...-->
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:xs="http
    <xsl:include href="file:///C:/Users/altova/Documents/Altova/MapForce2020/MapForceExamples
    <xsl:output method="xml" encoding="UTF-8" byte-order-mark="no" indent="yes"/>
    <xsl:template match="/">
        <CompletePO>
            <xsl:attribute name="xsi:noNamespaceSchemaLocation" namespace="http://www.w3.org/
CompletePO.xsd'"/>
            <xsl:for-each select="Customers/Customer">
                <Customer>
                    <Number>
                        <xsl:sequence select="xs:string(xs:integer(fn:string(Number)))"/>
                    </Number>
                    <FirstName>
                        <xsl:call-template name="tokenize">
                            <xsl:with-param name="string" select="FirstName" as="item()"/>
                        </xsl:call-template>
                    </FirstName>
                    <LastName>
                        <xsl:sequence select="fn:string(LastName)"/>
                    </LastName>
                </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.

To remove custom XSLT libraries from MapForce:

1. Click **Add/Remove Libraries** at the base of the Libraries window. The Manage Libraries window opens.
2. Click **Delete Library** next to the library that is to be deleted.

6.4.1.2 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\MapForce2023\MapForceExamples\Tutorial** folder:

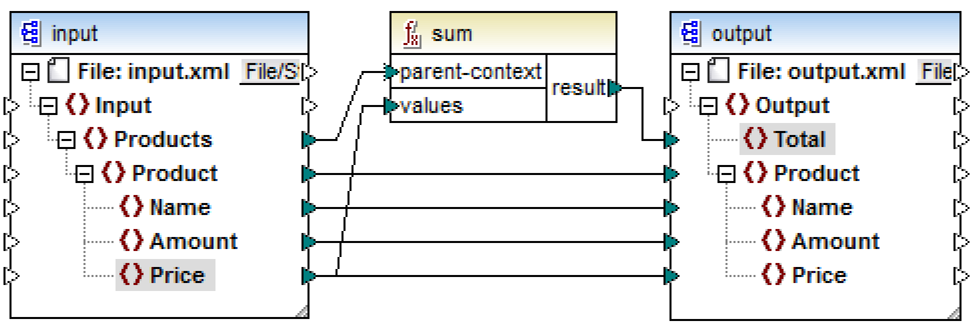
- **Summing-nodes.mfd** — the mapping file
- **input.xml** — the source XML file
- **input.xsd** — the source XML schema
- **output.xsd** — the target XML schema
- **Summing-nodes.xslt** — A custom XSLT stylesheet containing a named template to sum the individual nodes.

There are two different ways to achieve the goal of the mapping:

- By using the sum[234] function. This MapForce built-in function is available in the Libraries window.
- By importing a custom XSLT stylesheet into MapForce.

Solution 1: Using the "sum" aggregate function

To use the **`sum`** 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[227].

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>
```

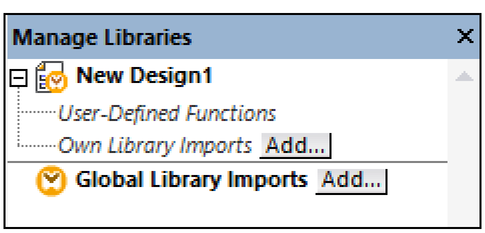
The code listing below shows a custom XSLT stylesheet which uses the named template "Total" and a single parameter `string`. The template works through the XML input file and sums all the values obtained by the XPath expression `/Product/Price`.

```
<?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"`.

Before importing the XSLT stylesheet into MapForce, select XSLT 1.0 as a transformation language[16]. You are now ready to import the custom function, as follows:

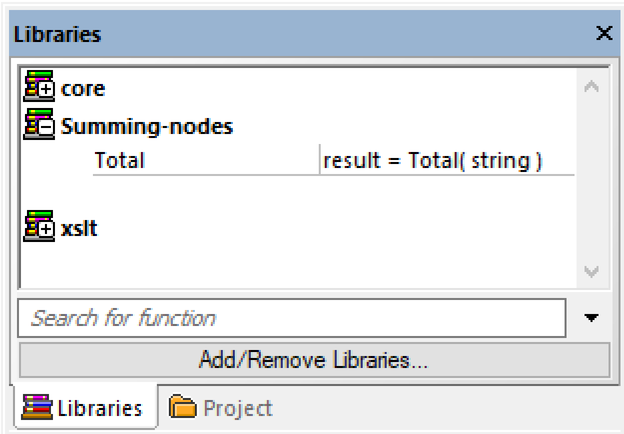
1. Click the **Add/Remove Libraries** button at the bottom of the Libraries window[22]. The **Manage Libraries** window opens (*see screenshot below*).



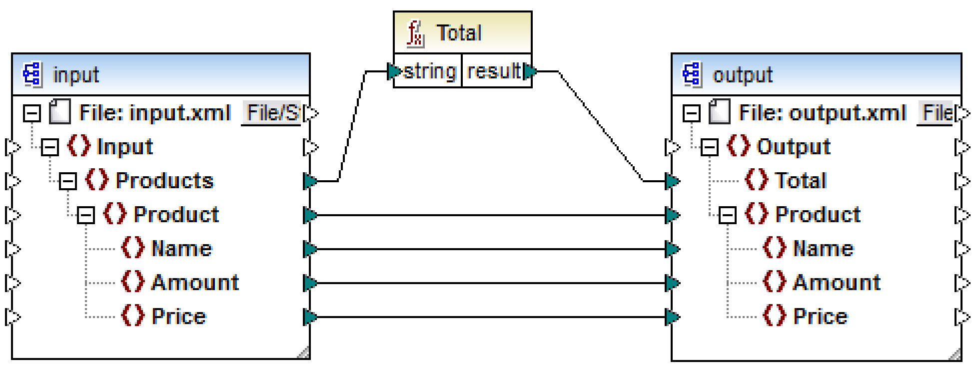
2. To import functions as a *local* library (in the scope of the current document only), click **Add** under the current mapping name. To import functions as a *global* library (at program level), click **Add** next to **Global Library Imports**. When you import a library *locally*, you can set the path of the library file to be relative to the mapping file. With globally imported libraries, the path of the imported library is always

absolute.

3. Browse for **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\Summing-nodes.xslt**, and click **Open**. A message box appears informing you that a new library has been added, and the new library appears in the Libraries window.



4. Drag the `Total` function from the Libraries 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>
```


6.5 Regular Expressions

When designing a MapForce mapping, you can use regular expressions ("regex") in the following contexts:

- In the **pattern** parameter of the **tokenize-regexp** [304] function.

The regular expression syntax and semantics for XSLT and XQuery are as defined in Appendix F of "XML Schema Part 2: Datatypes Second Edition".

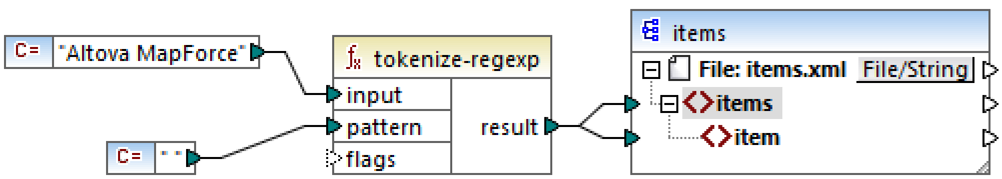
Note: When generating C++, C#, or Java code, the advanced features of the regular expression syntax might differ slightly. See the regex documentation of each language for more information.

Terminology

Let's examine the basic regular expression terminology by analyzing the **tokenize-regexp** function as an example. This function splits text into a sequence of strings, with the help of regular expressions. To achieve this, the function takes the following input parameters:

input	The input string to be processed by the function. The regular expression will operate on this string.
pattern	The actual regular expression pattern to be applied.
flags	This is an optional parameter that defines additional options (flags) that determine how the regular expression is interpreted, see "Flags" below.

In the mapping below, the input string is "Altova MapForce". The **pattern** parameter is a space character, and no regular expression flags are used.



This causes the text to be split whenever the space character occurs, so the mapping output is:

```
<items>
   <item>Altova</item>
   <item>MapForce</item>
</items>
```

Note that the **tokenize-regexp** function excludes the matched characters from the result. In other words, the space character in this example is omitted from the output.

The example above is very basic and the same result can be achieved without regular expressions, with the tokenize [303] function. In a more practical scenario, the **pattern** parameter would contain a more complex regular expression. The regular expression can consist of any of the following:

- Literals
- Character classes
- Character ranges
- Negated classes
- Meta characters
- Quantifiers

Literals

Use literals to match characters exactly as they are written (literally). For example, if input string is `abracadabra`, and **pattern** is the literal `br`, the output is:

```
<items>
   <item>a</item>
   <item>acada</item>
   <item>a</item>
</items>
```

The explanation is that the literal `br` had two matches in the input string `abracadabra`. After removing the matched characters from the output, the sequence of three strings illustrated above is produced.

Character classes

If you enclose a set of characters in square brackets (`[` and `]`), this creates a character class. One and only one of the characters inside the character class is matched, for example:

- The pattern `[aeiou]` matches any lowercase vowel.
- The pattern `[mj]ust` matches "must" and "just".

Note: The pattern is case sensitive, so a lowercase "a" does not match the uppercase "A". To make the matching case insensitive, use the **i** flag, see below.

Character ranges

Use `[a-z]` to create a range between the two characters. Only one of the characters will be matched at one time. For example, the pattern `[a-z]` matches any lowercase character between "a" and "z".

Negated classes

Using the caret (`^`) as the first character after the opening bracket negates the character class. For example, the pattern `[^a-z]` matches any character not in the character class, including newline characters.

Matching any character

Use the dot (`.`) meta character to match any single character, except for newline character. For example, the pattern `.` matches any single character.

Quantifiers

Within a regular expression, quantifiers define how many times the preceding character or sub-expression is allowed to occur in order for the match to take place.

?	Matches zero or one occurrences of the immediately preceding item. For example, the pattern `mo?` will match "m" and "mo".
+	Matches one or more occurrences of the immediately preceding item. For example, the pattern `mo+` will match "mo", "moo", "mooo", and so on.
*****	Matches zero or more occurrences of the immediately preceding item.
{min,max}	Matches any number of occurrences between *min* and *max*. For example, the pattern `mo{1,3}` matches "mo", "moo", and "mooo".

Parentheses

Parentheses `(` and `)` are used to group parts of a regex together. They can be used to apply quantifiers to a sub-expression (as opposed to just one character), or with alternation (see below).

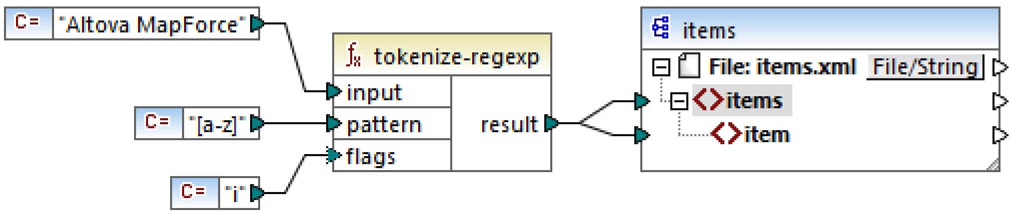
Alternation

The vertical bar (pipe) character `|` means "or". It can be used to match any of the several sub-expressions separated by `|`. For example, the pattern `(horse|make) sense` will match both "horse sense" and "make sense".

Flags

These are optional parameters that define how the regular expression is to be interpreted. Each flag corresponds to a letter. Letters may be in any order and can be repeated.

s	If this flag is present, the matching process operates in the "dot-all" mode. If the input string contains "hello" and "world" on two *different* lines, the regular expression `hello*world` will only match if the **s** flag is set.
m	If this flag is 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 this flag is present, the matching process operates in case-insensitive mode. For example, the regular expression `[a-z]` plus the **i** flag matches all letters a-z and A-Z.

	
x	If this flag is present, whitespace characters are removed from the regular expression prior to the matching process. Whitespace characters are **#x09**, **#x0A**, **#x0D** and **#x20**. **Note:** Whitespace characters within a character class are not removed, for example, `[#x20]`.

6.6 Function Library Reference

This reference section describes the MapForce built-in functions available in the Libraries window [22]. The functions are organized by library. The availability of function libraries in the **Libraries** window depends on the transformation language you choose for your mapping. To find out more about the list of available transformation languages, see this topic [16].

The information about the compatibility of functions and transformation languages is provided in the subsections below.

core functions

The lists below summarize the compatibility of core functions with transformation languages.
core | aggregate functions

- **avg**, **max**, **max-string**, **min**, **min-string**: XSLT 2.0, XSLT 3.0, XQuery 1.0, C#, C++, Java, Built-In;
- **count**, **sum**: all transformation languages.

core | conversion functions

- **boolean**, string, **number**: all transformation languages;
- **format-date**, **format-dateTime**, **format-time**: XSLT 2.0, XSLT 3.0, C#, C++, Java, Built-In;
- **format-number**: XSLT 1.0, XSLT 2.0, XSLT 3.0, C#, C++, Java, Built-In;
- **parse-date**, **parse-dateTime**, **parse-number**, **parse-time**: C#, C++, Java, Built-In.

core | file path functions
All the file path functions are compatible with all the transformation languages.

core | generator functions
The **auto-number** function is available for all the transformation languages.

core | logical functions
The logical functions are compatible with all the transformation languages.

core | math functions

- **add**, **ceiling**, **divide**, **floor**, **modulus**, **multiply**, **round**, **subtract**: all transformation languages;
- **round-precision**: C#, C++, Java, Built-In.

core | node functions

- **is-xsi-nil**, **local-name**, **static-node-annotation**, **static-node-name**: all transformation languages;
- **node-name**, **set-xsi-nil**, **substitute-missing-with-xsi-nil**: XSLT 2.0, XSLT 3.0, XQuery 1.0, C#, C++, Java, Built-In.

core | QName functions
The QName functions are compatible with all the transformation languages except for XSLT1.0.

core | sequence functions

- **exists**, **not-exists**, **position**, **substitute-missing**: all transformation languages;
- **distinct-values**, **first-items**, **generate-sequence**, **item-at**, **items-from-till**, **last-items**, **replicate-item**, **replicate-sequence**, **set-empty**, **skip-first-items**: XSLT 2.0, XSLT 3.0, XQuery 1.0, C#, C++, Java, Built-In;
- **group-adjacent**, **group-by**, **group-ending-with**, **group-into-blocks**, **group-starting-with**: XSLT 2.0, XSLT 3.0, C#, C++, Java, Built-In.

core | string functions

- **concat**, **contains**, **normalize-space**, **starts-with**, **string-length**, **substring**, **substring-after**, **substring-before**, **translate**: all transformation languages;
- **char-from-code**, **code-from-char**, **tokenize**, **tokenize-by-length**, **tokenize-regexp**: XSLT 2.0, XSLT 3.0, XQuery 1.0, C#, C++, Java, Built-In.

bson functions (MapForce Enterprise Edition only)

All the BSON functions are compatible only with Built-In.

db functions (MapForce Professional and Enterprise editions)

The `db` functions are compatible with C#, C++, Java, Built-In.

edifact functions (MapForce Enterprise Edition only)

The `edifact` functions are compatible with C#, C++, Java, Built-In.

lang functions (MapForce Professional and Enterprise editions)

The lists below summarize the compatibility of `lang` functions with transformation languages.
lang | datetime functions
The `lang | datetime` functions are compatible with C#, C++, Java, Built-In.

lang | file functions
The functions **read-binary-file** and **write-binary-file** are compatible only with Built-In.

lang | generator functions
The **create-guid** function is available for C#, C++, Java, Built-In.

lang | logical functions
The `lang | logical` functions are available for C#, C++, Java, Built-In.

lang | math functions
The `lang | math` functions are available for C#, C++, Java, Built-In.

lang | QName functions
The `lang | QName` functions are compatible with C#, C++, Java, Built-In.

lang | string functions

- **charset-decode**, **charset-encode**: Built-In;
- **match-pattern**: C#, Java, Built-In.

- **capitalize**, **count-substring**, **empty**, **find-substring**, **format-guid-string**, **left**, **left-trim**, **lowercase**, **pad-string-left**, **pad-string-right**, **repeat-string**, **replace**, **reversefind-substring**, **right**, **right-trim**, **string-compare**, **string-compare-ignore-case**, **uppercase**: C#, C++, Java, Built-In.

mime functions (MapForce Enterprise Edition only)

The `mime` functions are available for Built-In only.

xbrl functions (MapForce Enterprise Edition only)

The `xbrl` functions are compatible with C#, C++, Java, Built-In.

xlsx functions (MapForce Enterprise Edition only)

The `xlsx` functions are compatible with XSLT 2.0, XSLT 3.0, C#, Java, and Built-In.

xpath2 functions

All the `xpath2` functions are compatible with XSLT 2.0, XSLT 3.0, and XQuery 1.0.

xpath3 functions

All the `xpath3` functions are compatible only with XSLT 3.0.

xslt10 functions

The lists below summarize the compatibility of `xslt10` functions with transformation languages.
xslt10 | xpath functions

- **local-name**, **name**, **namespace-uri**: XSLT 1.0, XSLT 2.0, and XSLT 3.0.
- **lang**, **last**, **position**: XSLT 1.0.

xslt10 | xslt functions

- **generate-id**, **system-property**: XSLT 1.0, XSLT 2.0, and XSLT 3.0.
- **current**, **document**, **element-available**, **function-available**, **unparsed-entity-uri**: XSLT 1.0.

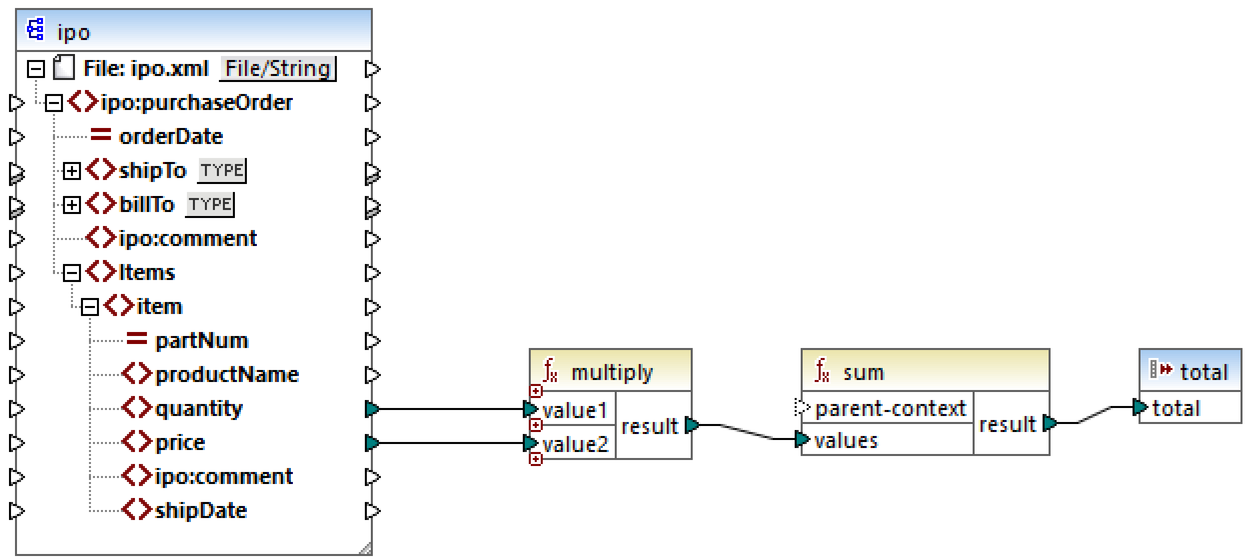
6.6.1 core | aggregate functions

"Aggregating" means processing multiple values of the same type so as to obtain a single result, such as a sum, a count, or an average. You can perform data aggregation in MapForce with the help of aggregation functions, such as `avg`, `count`, `max`, and others.

The following two arguments are common to all aggregation functions:

1. **parent-context**. This argument is optional; it lets you override the default mapping context (and thus change the scope of the function, or the values that the function must iterate over). For a worked example, see Example: Changing the Parent Context 404.

2. **values**. This argument must be connected to a source item that supplies the values to be processed. For example, in the mapping illustrated below, the **sum** function takes as input a sequence of numeric values that originates from a source XML file. For each item in the source XML file, the **multiply** function gets the item's price times quantity, and passes the result to the **sum** function. The **sum** function will aggregate all input values and produce a total result that is also the output of the mapping. You can find this mapping in the **MapForceExamples** folder.

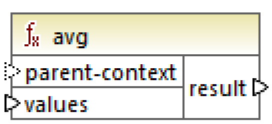


SimpleTotal.mfd

Some aggregate functions, such as `min`, `max`, `sum`, and `avg`, work exclusively with numeric values. The input data of these functions is converted to the **decimal** data type for processing.

6.6.1.1 avg

Returns the average value of all values within the input sequence. The average of an empty set is an empty set.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

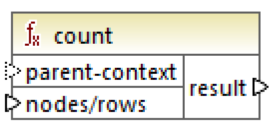
Argument	Description
parent-context	Optional argument. Supplies the parent context. See also Example: Changing the Parent Context[404]. The `parent-context` argument is an optional argument in some MapForce core aggregation functions (e.g., **`min`**, **`max`**, **`avg`**, **`count`**). In a source component which has multiple hierarchical sequences, the parent context determines the set of nodes on which the function should operate.
values	This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric.

Example

See Example: Grouping Records by Key[187].

6.6.1.2 count

Returns the number of individual items making up the input sequence. The count of an empty set is zero.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Note that this function has limited functionality in XSLT 1.0.

Parameters

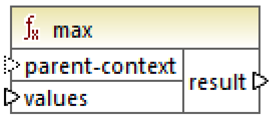
Argument	Description
parent-context	Optional argument. Supplies the parent context. See also Example: Changing the Parent Context[404]. The `parent-context` argument is an optional argument in some MapForce core aggregation functions (e.g., **`min`**, **`max`**, **`avg`**, **`count`**). In a source component which has multiple hierarchical sequences, the parent context determines the set of nodes on which the function should operate.
nodes/rows	This argument must be connected to the source item to be counted.

Example

See Example: Changing the Parent Context[404].

6.6.1.3 max

Returns the maximum value of all numeric values in the input sequence. The maximum of an empty set is an empty set.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

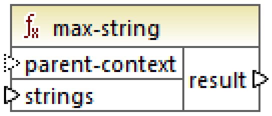
Argument	Description
parent-context	Optional argument. Supplies the parent context. See also Example: Changing the Parent Context[404]. The `parent-context` argument is an optional argument in some MapForce core aggregation functions (e.g., **`min`**, **`max`**, **`avg`**, **`count`**). In a source component which has multiple hierarchical sequences, the parent context determines the set of nodes on which the function should operate.
values	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 max-string[230] function.

Example

See Example: Grouping Records by Key[187].

6.6.1.4 max-string

Returns the maximum value of all string values in the input sequence. For example, **`max-string("a", "b", "c")`** returns **`"c"`**. The function returns an empty set if the **strings** argument is an empty set.



Languages

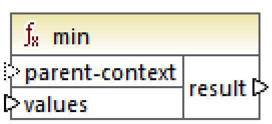
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
parent-context	Optional argument. Supplies the parent context. See also Example: Changing the Parent Context[404]. The `parent-context` argument is an optional argument in some MapForce core aggregation functions (e.g., **`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.
strings	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 `xs:string`.

6.6.1.5 min

Returns the minimum value of all numeric values in the input sequence. The minimum of an empty set is an empty set.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
parent-context	Optional argument. Supplies the parent context. See also Example: Changing the Parent Context[404]. The `parent-context` argument is an optional argument in some MapForce core

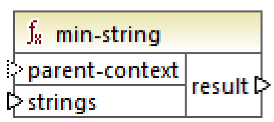
Argument	Description
	aggregation functions (e.g., **min**, **max**, **avg**, **count**). In a source component which has multiple hierarchical sequences, the parent context determines the set of nodes on which the function should operate.
values	This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric. To get the minimum from a sequence of strings, use the min-string [232] function.

Example

See Example: Grouping Records by Key [187].

6.6.1.6 min-string

Returns the minimum value of all string values in the input sequence. For example, `min-string("a", "b", "c")` returns `"a"`. The function returns an empty set if the **strings** argument is an empty set.



Languages

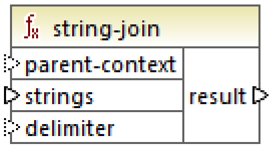
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
parent-context	Optional argument. Supplies the parent context. See also Example: Changing the Parent Context [404]. The `parent-context` argument is an optional argument in some MapForce core aggregation functions (e.g., **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.
strings	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 `xs:string`.

6.6.1.7 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 function returns an empty string if the **strings** argument is an empty set.



Languages

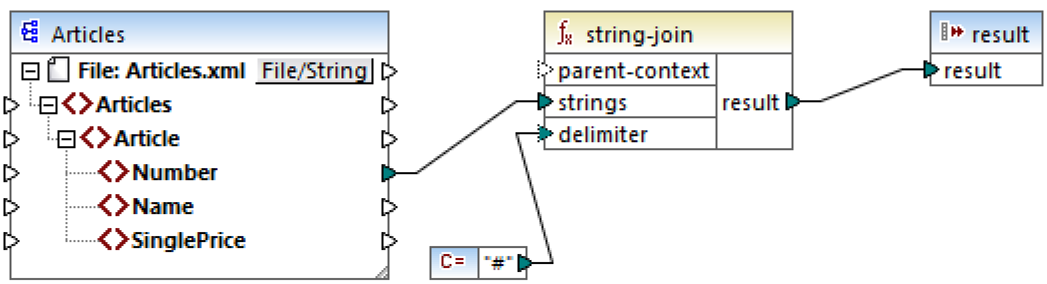
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
parent-context	Optional argument. Supplies the parent context. See also Example: Changing the Parent Context (404). The `parent-context` argument is an optional argument in some MapForce core aggregation functions (e.g., **`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.
strings	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 `xs:string`.
delimiter	Optional argument. Specifies the delimiter to be inserted between any two consecutive strings.

Example

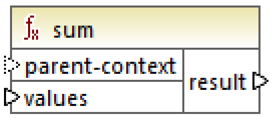
In the example below, the source XML file contains four **Article** items, with the following numbers: 1, 2, 3, and 4.



The constant supplies the character "#" as the delimiter. The mapping result is, therefore, **1#2#3#4**. If you do not supply a delimiter, then the result becomes **1234**.

6.6.1.8 sum

Returns the arithmetic sum of all values in the input sequence. The sum of an empty set is zero.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
parent-context	Optional argument. Supplies the parent context. See also Example: Changing the Parent Context [404]. The `parent-context` argument is an optional argument in some MapForce core aggregation functions (e.g., **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.
values	This argument must be connected to a source item which supplies the actual data. Note that the supplied argument value must be numeric.

Example

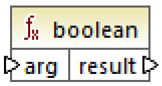
See Example: Summing Node Values [217].

6.6.2 core | conversion functions

To support explicit data type conversion, several type conversion functions are available in the **conversion** library. Note that the conversion functions are not always necessary because, in most cases, MapForce creates the necessary conversions automatically. Conversion functions are typically useful to format date and time values, or to compare values. For example, if some mapping items are of differing types (such as integer and string), you can use the number [243] conversion function to force a numeric comparison.

6.6.2.1 boolean

Converts the value of **arg** to a Boolean value. This may be useful for working with logical functions (such as **`equal`**, **`greater`**, and so on), as well as filters and if-else conditions[168]. To get a Boolean **false**, supply an empty string or numeric 0 as argument. To get a Boolean **true**, supply a non-empty string or numeric 1 as argument.



Languages

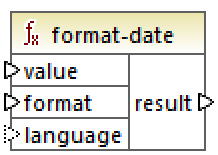
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameter

Argument	Description
arg	Mandatory argument. Supplies the value to be converted.

6.6.2.2 format-date

Converts a date value of type `xs:date` to a string and formats it according to specified options.



Languages

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

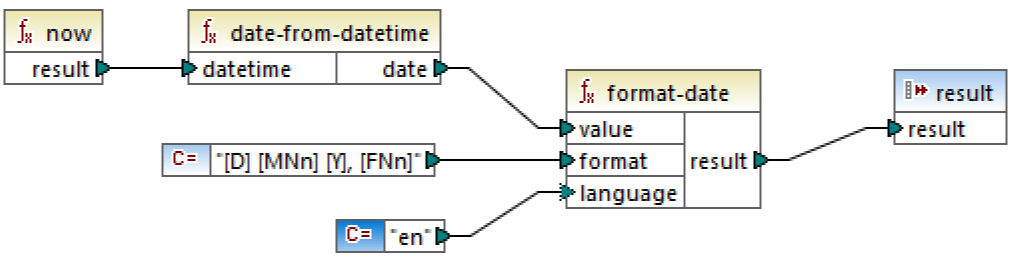
Parameters

Argument	Description
value	The `xs:date` value to be formatted.
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 **format** argument in the format-dateTime[236] 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:

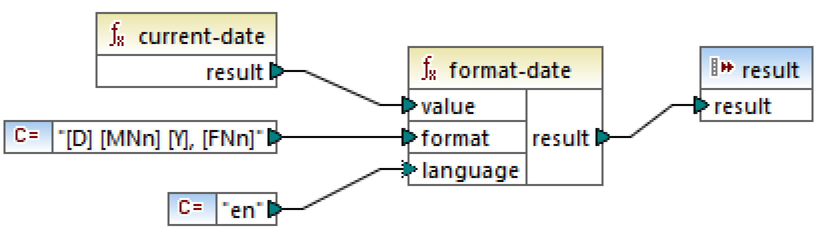
Argument	Description
	de German **en (default)** English **es** Spanish **fr** French **ja** Japanese

Example

The following mapping outputs the current date in a format like: "25 March 2020, Wednesday". To translate this value to Spanish, set the value of the **language** argument to **es**.

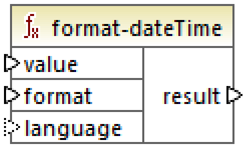


Note that the mapping above is designed for the Built-in, C++, C#, or Java transformation languages. In XSLT 2.0, the same result can be achieved by the following mapping:



6.6.2.3 format-dateTime

Converts a value of type `xs:dateTime` to a string. The string representation of date and time is formatted according to the value of the **format** argument.



Languages

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
value	The `xs:dateTime` value to be formatted.
format	A format string identifying the way in which **value** is to be formatted. See "Remarks" below.
language	Optional argument. When supplied, the name of the month and the day of the week are returned in a specific language. Valid values: **de** German **en (default)** English **es** Spanish **fr** French **ja** Japanese

Note: If the function's output (result) is connected to an item of type other than string, the formatting may be lost as the value is cast to the target type. To disable this automatic cast, clear the **Cast target values to target types** check box in the Component Settings 71 of the target component.

Remarks

The **format** argument consists of a string containing so-called variable markers enclosed in square brackets, for example **`[Y]/[M]/[D]`**. Characters outside the square brackets are literal characters. 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
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
01	Decimal format, two digits	01, 02, 03
N	Name of component, upper case[1]	MONDAY, TUESDAY
n	Name of component, lower case[1]	monday, tuesday
Nn	Name of component, title case[1]	Monday, Tuesday

Footnotes:

1. The **N**, **n**, and **Nn** modifiers are supported by the following components only: **M**, **d**, **D**.

The width modifier, if necessary, is introduced by a comma, followed by a digit that expresses the minimum width. Optionally, you can add a dash followed by another digit that expresses the maximum width. For example:

- **`[D,2]`** is the day of the month, with leading zeros (two digits).
- **`[MNn,3-3]`** is the name of the month, written as three characters, e.g. *Jan*, *Feb*, *Mar*, and so on.

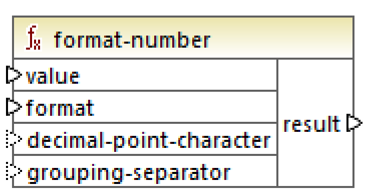
Examples

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
2010-06-02T08:02	`[Y] [MNn] [D] [F,3-3] [H01]:[m]:[s]`	2010 June 2 Wed 08:02:12

6.6.2.4 format-number

Converts a number into a string and formats it according to the specified options.



Languages

Built-in, C++, C#, Java, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

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. See "Remarks" below.
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 (result) is connected to an item of type other than string, the formatting may be lost as the value is cast to the target type. To disable this automatic cast, clear the **Cast target values to target types** check box in the Component Settings 71 of the target component.

Remarks

The **format** argument takes the following form:

```
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 table below illustrates examples of format strings and their result.

Note: The rounding method used by the **format-number** function is "half up", which means that 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 applies only to generated program 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".

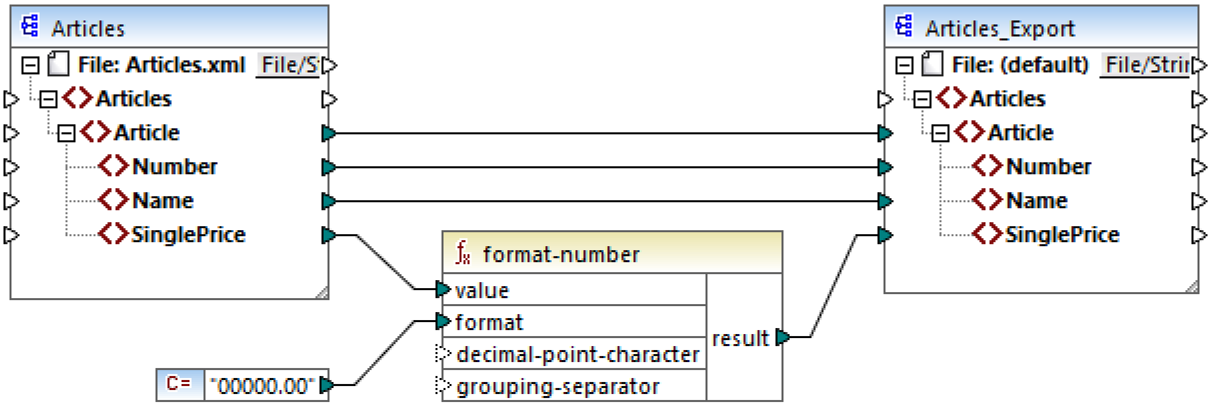
Number	Format String	Result
1234.5	#,##0.00	1,234.50
123.456	#,##0.00	123.46
1000000	#,##0.00	1,000,000.00
-59	#,##0.00	-59.00
1234	###0.0###	1234.0
1234.5	###0.0###	1234.5
.00025	###0.0###	0.0003
.00035	###0.0###	0.0004
0.25	#00%	25%
0.736	#00%	74%
1	#00%	100%
-42	#00%	-4200%
-3.12	#.00;(#.00)	(3.12)
-3.12	#.00;#.00CR	3.12CR

Example

The mapping illustrated below reads data from source XML and writes it to a target XML. There are multiple **SinglePrice** elements in the source that contain the following decimal values: **25**, **2.30**, **34**, **57.50**. The mapping has two goals:

1. Pad all values with zeros to the left so that the significant part takes 5 digits exactly
2. Pad all values with zeros to the right so that the decimal part takes 2 digits exactly

To achieve this, the format string `00000.00` was supplied as argument to the **format-number** function.



PreserveFormatting.mfd

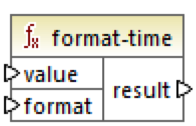
Consequently, the values in the target have become:

```
00025.00
00002.30
00034.00
00057.50
```

You can find the mapping design file at the following path: **<Documents>\Altova\MapForce2023\MapForceExamples\PreserveFormatting.mfd**.

6.6.2.5 format-time

Converts an `xs:time` input value into a string.



Languages

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

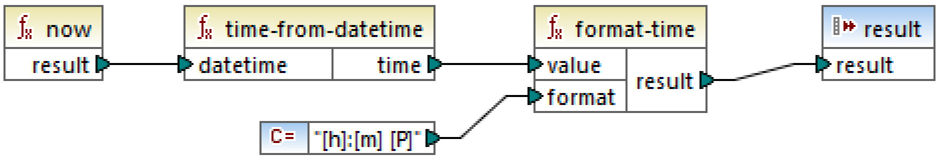
Parameters

Argument	Description
value	Mandatory argument. Supplies the `xs:time` value to be formatted.
format	Mandatory argument. Supplies a format string. This argument is used in the same way as the **format** argument in the format-dateTime 236 function.

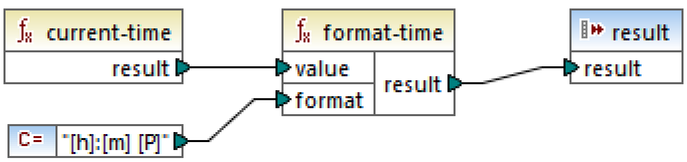
Example

The following mapping outputs the current time in a format like `2:15 p.m.` . To achieve this, it uses the format string `[h]:[m] [P]`, where:

- **[h]** is the current hour in 12-hour format
- **[m]** is the current minute
- **[P]** is the "a.m." or "p.m." part

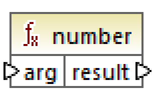


Note that the mapping above is designed for the Built-in, C++, C#, or Java transformation languages. In XSLT 2.0, the same result can be achieved by the following mapping:



6.6.2.6 number

Converts the value of **arg** into a number, where **arg** is a string or Boolean value. If **arg** is a string, MapForce will attempt to parse it as a number. For example, a string like `"12.56"` is converted to the decimal value `12.56`. If **arg** is Boolean **true**, it is converted to numeric **1**. If **arg** is Boolean **false**, it is converted to numeric **0**.



Languages

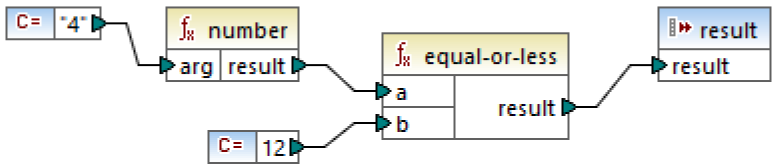
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
arg	Mandatory argument. Supplies the value to be converted.

Example

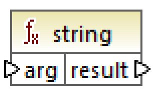
In the example below, the first constant is of type `string` and it contains the string "4". The second constant contains the numeric constant 12. In order for the two values to be compared as numbers, the types must agree.



Adding a **`number`** function to the first constant converts the string "4" to the numeric value of 4. The result of the comparison is then "true". If the **`number`** function were not used (that is, if "4" was connected directly to **a**), a string comparison would occur, with the result being "false".

6.6.2.7 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 an XML complex type, then all descendants are also output as a single string.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

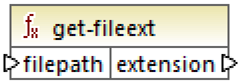
Argument	Description
arg	Mandatory argument. Supplies the value to be converted.

6.6.3 core | file path functions

The **file path** functions allow you to directly access and manipulate file path data, such as folders, file names, and extensions for further processing in your mappings. They can be used in all languages supported by MapForce.

6.6.3.1 get-fileext

Returns the extension of the file path including the dot "." character.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

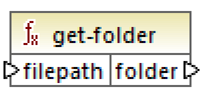
Argument	Description
filepath	Mandatory argument. Supplies the file path to be processed.

Example

If you supply "c:\data\Sample.mfd" as argument, the result is `.mfd`.

6.6.3.2 get-folder

Returns the folder name of the file path including the trailing slash, or backslash character.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

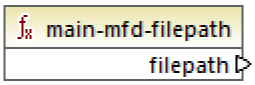
Argument	Description
filepath	Mandatory argument. Supplies the file path to be processed.

Example

If you supply "c:\data\Sample.mfd" as argument, the result is `c:\data\`.

6.6.3.3 main-mfd-filepath

Returns the full path of the mapping design file (.mfd) containing the main mapping. An empty string is returned if the .mfd is currently not saved.

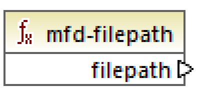


Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

6.6.3.4 mfd-filepath

If the function is called in the main mapping, it returns the same as the main-mfd-filepath [246] function, i.e. the full path of the .mfd file containing the main mapping. An empty string is returned if the .mfd file is currently not saved. If called within a user-defined function which is *imported* by an .mfd file, it returns the full path of the *imported* .mfd file that contains the definition of the user-defined function.

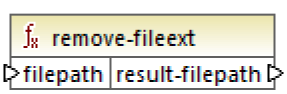


Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

6.6.3.5 remove-fileext

Removes the extension of the file path, including the dot character.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

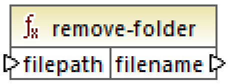
Argument	Description
filepath	Mandatory argument. Supplies the file path to be processed.

Example

If you supply "c:\data\Sample.mfd" as argument, the result is `c:\data\Sample`.

6.6.3.6 remove-folder

Removes the directory of the file path, including the trailing slash, or backslash character.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

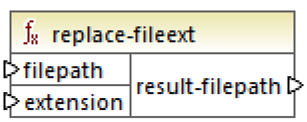
Argument	Description
filepath	Mandatory argument. Supplies the file path to be processed.

Example

If you supply "c:\data\Sample.mfd" as argument, the result is `Sample.mfd`.

6.6.3.7 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.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

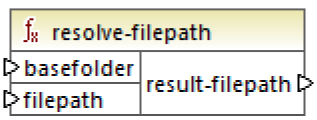
Argument	Description
filepath	Mandatory argument. Supplies the file path to be processed.
extension	Mandatory argument. Supplies the new extension to use.

Example

If you supply "c:\data\Sample.log" as **filepath**, and ".txt" as **extension**, the result is `c:\data\Sample.txt`.

6.6.3.8 resolve-filepath

Resolves a relative file path against a base folder. The function supports '.' (current directory) and '..' (parent directory).



Languages

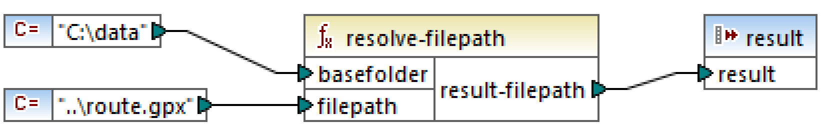
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
basefolder	Mandatory argument. Supplies the base directory relative to which the path should be resolved. This can be an absolute or relative path.
filepath	Mandatory argument. Supplies the relative file path to be resolved.

Examples

In the mapping below, the relative file path **..\route.gpx** is resolved against the **C:\data** directory.



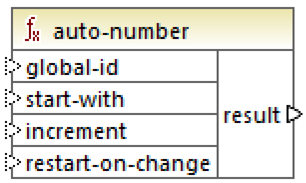
The mapping result is `C:\route.gpx`.

6.6.4 core | generator functions

The **core / generator** functions library includes functions which generate values.

6.6.4.1 auto-number

Generates integer numbers in a sequence (for example, 1,2,3,4, ...). It is possible to set the starting integer, the increment value, and other options by means of parameters.



> The exact order in which functions are called by the generated mapping code is undefined. MapForce may need to cache calculated results for reuse, or evaluate expressions in any order. Also, unlike other functions, the `auto-number` function returns a different result when called multiple times with the same input parameters. Therefore, it is strongly recommended to use the `auto-number` function cautiously. In some cases, it is possible to achieve the same result by using the position [288] function instead.

Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

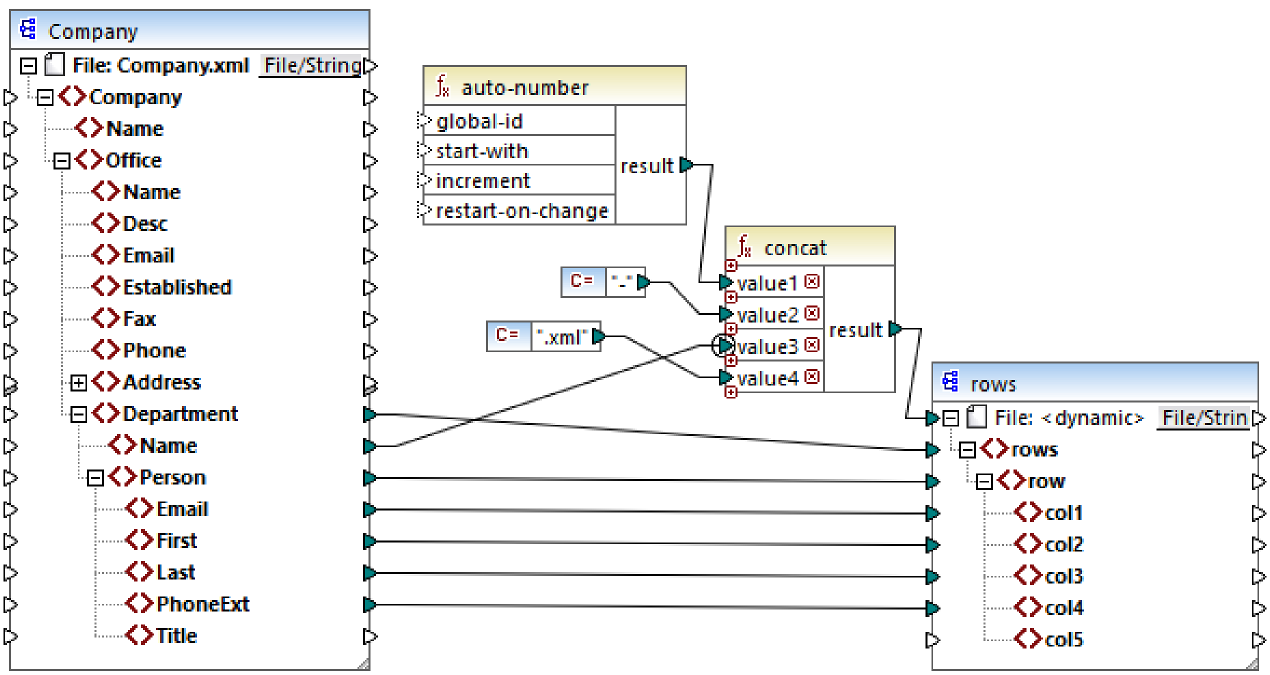
Parameters

Argument	Description
global-id	Optional parameter. If a mapping design contains multiple `auto-number` functions, they will generate sequences with duplicate (overlapping) numbers. To make all `auto-number` functions aware of each other, and thus generate sequences that do not overlap, connect a common string (for example, a constant) to the **global-id** input of each `auto-number` function.
start-with	Optional parameter. Specifies the integer with which the generated sequence begins. The default value is **1**.
increment	Optional parameter. Specifies the increment value. The default value is **1**.
restart-on-change	Optional parameter. Resets the counter to **start-with**, when the content of the connected item changes.

Example

The following mapping is a variation of the **ParentContext.mfd** mapping discussed in the [Example: Changing the Parent Context](404).

The goal of the mapping illustrated below is to generate multiple XML files, one for each department in the source XML file. There are some departments with the same name (that's because they belong to different parent offices). For this reason, each generated file name must begin with a sequential number, for example **1-Administration.xml**, **2-Marketing.xml**, and so on.



To achieve the mapping goal, the **`auto-number`** function was used. The result of this function is concatenated with a dash character, followed by the department name, followed by the ".xml" string in order to create the unique name of the generated file. Importantly, the third parameter of the **`concat`** function (the department name) has a [priority context](408) applied. This has the effect that the **`auto-number`** function is called in the context of each department, and produces the required sequential values. If priority context were not used, the **`auto-number`** function would keep generating number 1 (in the absence of any context), and duplicate file names would be generated as a consequence.

6.6.5 core | logical functions

Logical functions are (generally) used to compare input data and return a Boolean `true` or `false`. They are generally used to test data before passing on a subset to the target component using a [filter](168). Nearly all logical functions have the following structure:

- input parameters: `a | b` or `value1 | value2`
- output parameter: `result`

The evaluation result 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 parameters contain string values **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**).

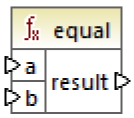
If all input values are of the same data type, then the comparison is done for the common type. If input values are of different types (for example, `integer` and `string`, or `string` and `date`), then the data type used for the comparison is the most general (least restrictive) of the two.

Before comparing two values of different types, all input values are converted to a common data type. Using the previous example, the data type `string` is less restrictive than `integer`. Comparing the integer value **4** with the string **12** converts the 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 an argument to a logical function, it returns a null value. For more information about handling null values, see Nil Values / Nillable 113.

6.6.5.1 equal

The **equal** function (*see screenshot below*) returns Boolean `true` if **a** is the same as **b**; `false` otherwise. The comparison is case-sensitive.



Example:

```
a = hi
b = hi
```

In this example, both values are the same. Therefore, the result is `true`. If, for instance, `b` equaled `Hi`, the function would return `false`.

Languages

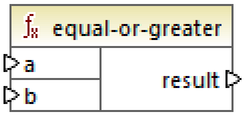
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
a	Mandatory parameter. Provides the first value to compare.
b	Mandatory parameter. Provides the second value to compare.

6.6.5.2 equal-or-greater

Returns Boolean **true** if *a* is equal to or greater than *b*; **false** otherwise.



Languages

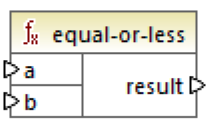
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
a	Mandatory parameter. Provides the first value to compare.
b	Mandatory parameter. Provides the second value to compare.

6.6.5.3 equal-or-less

Returns Boolean **true** if *a* is equal to or less than *b*; **false** otherwise.



Languages

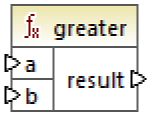
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
a	Mandatory parameter. Provides the first value to compare.
b	Mandatory parameter. Provides the second value to compare.

6.6.5.4 greater

Returns Boolean **true** if *a* is greater than *b*; **false** otherwise.



Languages

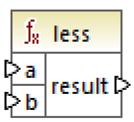
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
a	Mandatory parameter. Provides the first value to compare.
b	Mandatory parameter. Provides the second value to compare.

6.6.5.5 less

Returns Boolean **true** if *a* is less than *b*; **false** otherwise.



Languages

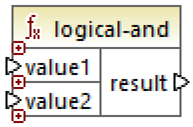
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
a	Mandatory parameter. Provides the first value to compare.
b	Mandatory parameter. Provides the second value to compare.

6.6.5.6 logical-and

Returns Boolean **true** only if each input value is true; **false** otherwise. You can connect the result to another **logical-and** function and thus join an arbitrary number of conditions with logical AND, in order to test that they all return **true**. Also, this function can be extended to take additional arguments, see [Add or Delete Function Arguments](192).



Languages

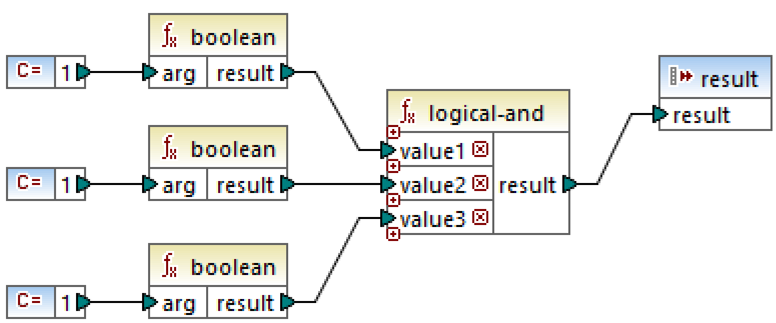
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
value1	Mandatory parameter. Provides the first value to compare.
value2	Mandatory parameter. Provides the second value to compare.

Example

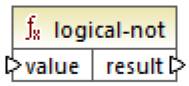
The mapping illustrated below returns **true** because all input values to the **logical-and** function are **true** as well. If any of the input values were **false**, then the mapping's result would be **false** as well.



See also [Example: Look-up and Concatenation](210).

6.6.5.7 logical-not

Inverts or flips the logical result of the input value. For example, if *value* is **true**, the function's result is false. If *value* is **false**, then result is true.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

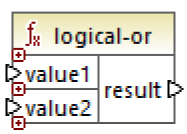
Parameters

Argument	Description
value	Mandatory parameter. Provides the input value.

6.6.5.8 logical-or

This function requires both input values to be Boolean. If at least one of the input values is **true**, then the result is **true**. Otherwise, the result is **false**.

This function can be extended to take additional arguments, see Add or Delete Function Arguments[192].



Languages

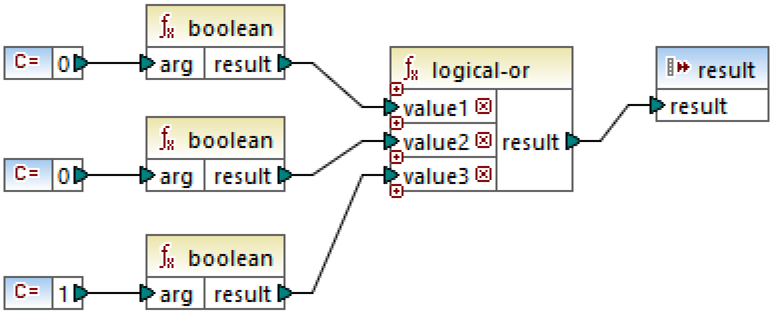
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
value1	Mandatory parameter. Provides the first value to compare.
value2	Mandatory parameter. Provides the second value to compare.

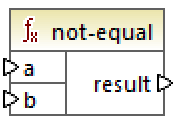
Example

The result of the mapping below is **true**, because at least one of the function's arguments is **true**.



6.6.5.9 not-equal

Returns Boolean **true** if *a* is not equal to *b*; **false** otherwise.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

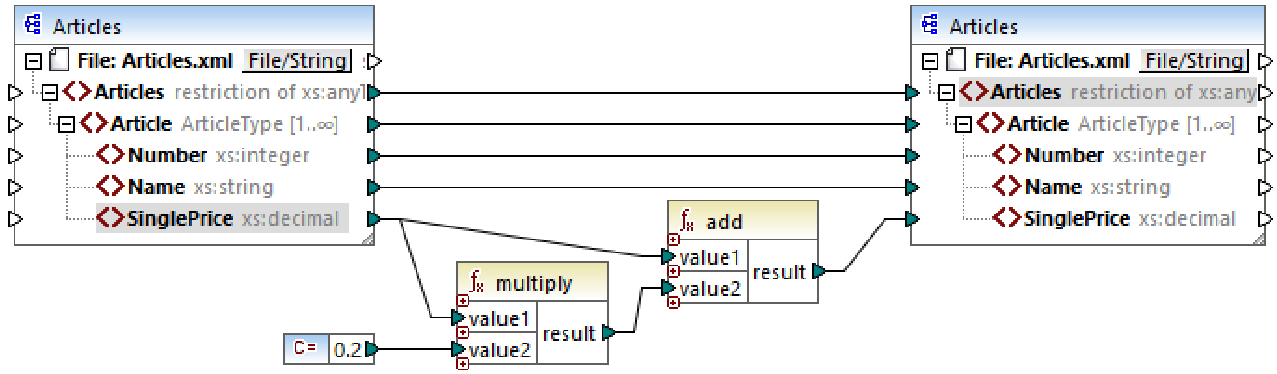
Parameters

Argument	Description
a	Mandatory parameter. Provides the first value to compare.
b	Mandatory parameter. Provides the second value to compare.

6.6.6 core | math functions

Math functions are used to perform basic mathematical operations on data. Note that they cannot be used to perform computations on durations or `datetime` values.

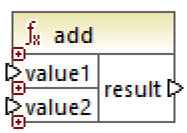
Most math functions take two input parameters (**value1**, **value2**) that are operands of the mathematical operation. The input values are automatically converted to `decimal` type for further processing. The result of math functions is also of `decimal` type.



The example shown above adds 20% sales tax to each of the articles mapped to the target component.

6.6.6.1 add

Adds **value1** to **value2** and returns the result as a decimal value. This function can be extended to take additional arguments, see Add or Delete Function Arguments 192.



Languages

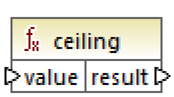
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
value1	Mandatory parameter. Provides the first operand.
value2	Mandatory parameter. Provides the second operand.

6.6.6.2 ceiling

Returns the smallest integer that is greater than or equal to **value**.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

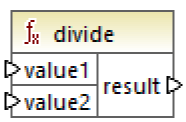
Argument	Description
value	Mandatory parameter. Provides the function's input value.

Example

If the input value is **11.2**, then applying the `ceiling` function to it makes the result **12**, i.e. the smallest integer that is greater than **11.2**.

6.6.6.3 divide

Divides **value1** by **value2** and returns the result as decimal value. The result precision depends on the target language. Use the round-precision [261] function to define the precision of result.



Languages

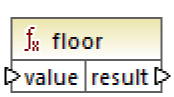
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
value1	Mandatory parameter. Provides the first operand.
value2	Mandatory parameter. Provides the second operand.

6.6.6.4 floor

Returns the greatest integer that is less than or equal to **value**.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

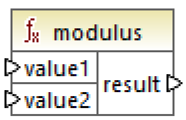
Argument	Description
value	Mandatory parameter. Provides the function's input value.

Example

If the input value is **11.7**, then applying the `floor` function to it makes the result **11**, i.e. the greatest integer than is less than **11.7**.

6.6.6.5 modulus

Returns the remainder of dividing **value1** by **value2**.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
value1	Mandatory parameter. Provides the first operand.
value2	Mandatory parameter. Provides the second operand.

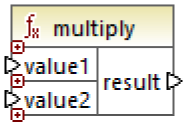
Example

If the input values are **1.5** and **1**, then the result of the `modulus` function is **0.5**. The explanation is that `1.5 / 1` leaves a remainder of **0.5**.

If the input values are **9** and **3**, then the result is **0**, since `9 / 3` leaves no remainder.

6.6.6.6 multiply

Multiplies **value1** by **value2** and returns the result as a decimal value.



Languages

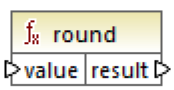
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
value1	Mandatory parameter. Provides the first operand.
value2	Mandatory parameter. Provides the second operand.

6.6.6.7 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".



Languages

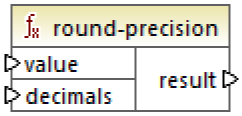
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
value	Mandatory parameter. Provides the function's input value.

6.6.6.8 round-precision

Rounds the input value to *N* decimal places, where *N* is the **decimals** argument.



Languages

Built-in, C++, C#, Java.

Parameters

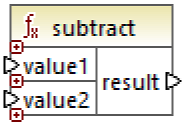
Argument	Description
value	Mandatory parameter. Provides the function's input value.
decimals	Mandatory parameter. Specifies the number of decimals to round to.

Example

Rounding the value **2.777777** to 2 decimals yields **2.78**. Rounding the value **0.1234** to 3 decimals yields **0.123**.

6.6.6.9 subtract

Subtracts **value2** from **value1** and returns the result as decimal value.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
value1	Mandatory parameter. Provides the first operand.
value2	Mandatory parameter. Provides the second operand.

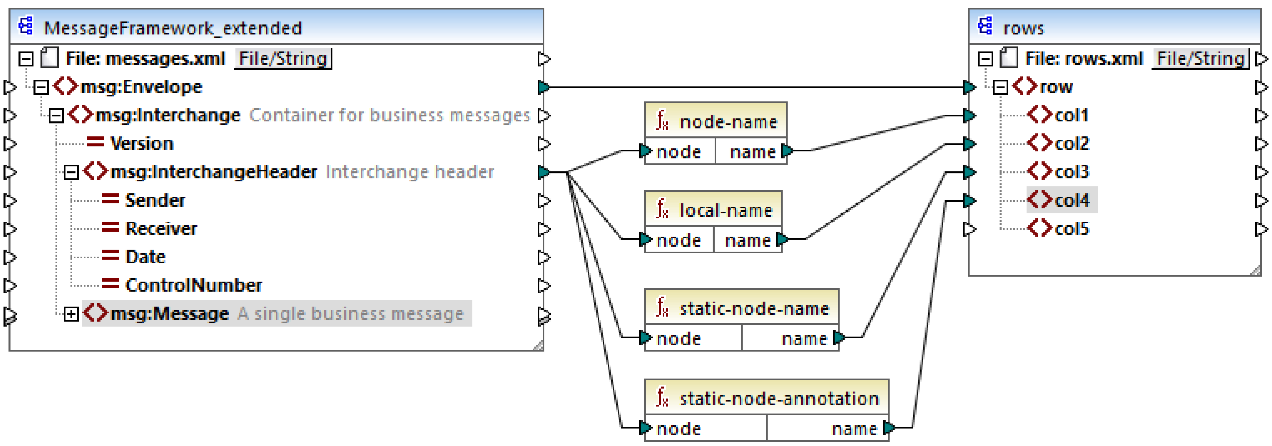
6.6.7 core | node functions

The functions from the **core | node functions** library allow you to access information about nodes on a mapping component (such as the node name or annotation), or to process nillable elements, see also Nil Values / Nillable 113.

Be aware that there is an alternative way to access node names, which does not require node functions at all, see Mapping Node Names 380.

The mapping illustrated below shows a few node functions that get information from the **msg:InterchangeHeader** node of the source XML file. More specifically, the following information is extracted:

1. The **node-name** function returns the qualified name of the node, which includes the node prefix.
2. The **local-name** function returns just the local part.
3. The **static-node-name** function is similar to the **node-name** function, but is available in XSLT 1.0 as well.
4. The **static-node-annotation** function gets the element's annotation as it was defined in the XML schema.

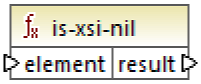


The output of the mapping is as follows (excluding the XML and namespace declarations):

```
<row>
   <col1>msg:InterchangeHeader</col1>
   <col2>InterchangeHeader</col2>
   <col3>msg:InterchangeHeader</col3>
   <col4>Interchange header</col4>
</row>
```

6.6.7.1 is-xsi-nil

Returns **true** if the **element** node has the `xsi:nil` attribute set to **true**.



Languages

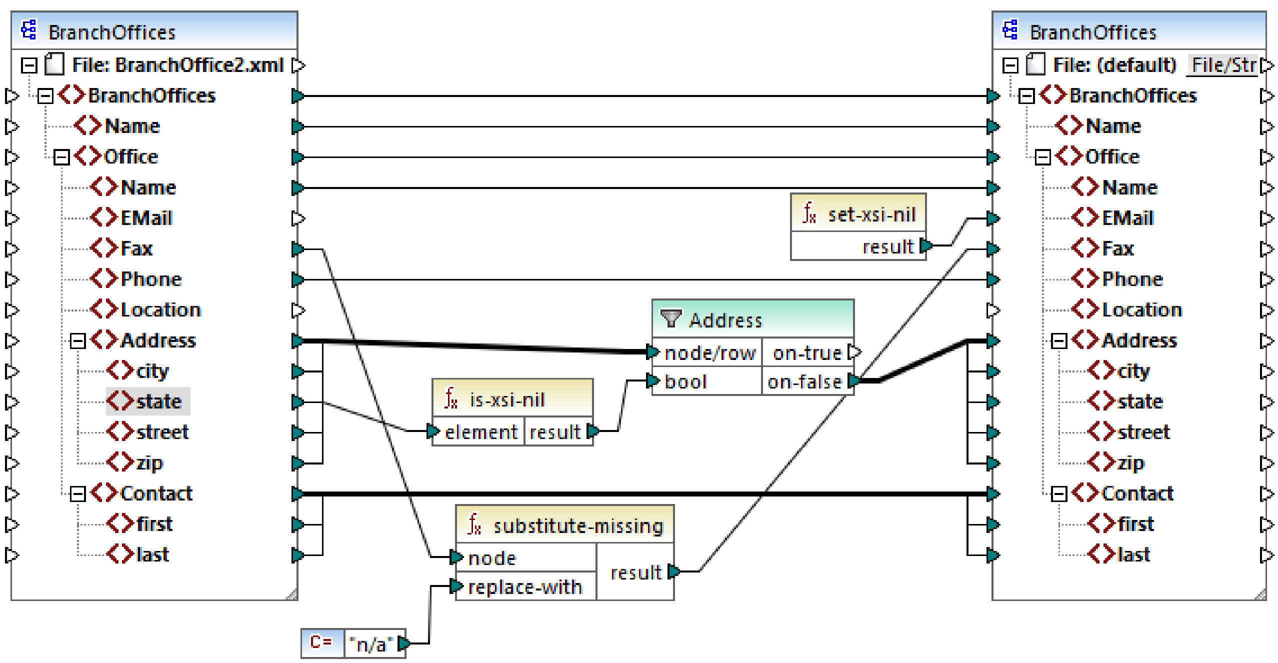
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
element	Mandatory parameter. Must be connected to the source node that is to be checked.

Example

The mapping design illustrated below copies data from a source to a target XML file conditionally, and also illustrates the usage of several functions, including **`is-xsi-nil`**. This mapping is called **HandlingXsiNil.mfd** and can be found in the **<Documents>\Altova\MapForce2023\MapForceExamples** directory.



As illustrated above, the **is-xsi-nil** function checks whether the `xsi:nil` attribute is "true" for the **state** item in the source file. If this attribute is "false", the filter will copy the parent **Address** element to the target. The source XML file looks as follows (excluding the XML and namespace declarations):

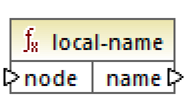
```xml
<BranchOffices>
   <Name>Nanonull</Name>
   <Office>
      <Name>Nanonull Research Outpost</Name>
      <EMail>sp@nanonull.com</EMail>
      <Fax xsi:nil="true"/>
      <Phone>+8817 3141 5926</Phone>
      <Address>
         <city>South Pole</city>
         <state xsi:nil="true"/>
         <street xsi:nil="true"/>
         <zip xsi:nil="true"/>
      </Address>
      <Contact>
         <first>Scott</first>
         <last>Amundsen</last>
      </Contact>
   </Office>
</BranchOffices>
```

The result of the mapping is that no **Address** is copied to the target at all, because there is only one **Address** in the source, and the `xsi:nil` attribute is set to "true" for the **state** element. Consequently, the mapping output is as follows:

```xml
<BranchOffices>
   <Name>Nanonull</Name>
   <Office>
      <Name>Nanonull Research Outpost</Name>
      <EMail xsi:nil="true"/>
      <Fax>n/a</Fax>
      <Phone>+8817 3141 5926</Phone>
      <Contact>
         <first>Scott</first>
         <last>Amundsen</last>
      </Contact>
   </Office>
</BranchOffices>
```

6.6.7.2 local-name

Returns the local name of the node. Unlike the node-name[265] function, **local-name** does not return the node's prefix. If the node does not have a prefix, then **local-name** and **node-name** return the same value.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

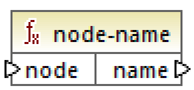
Parameters

Argument	Description
node	Mandatory parameter. Connect this input to the node whose name you want to get.

6.6.7.3 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 works only on those nodes that have a name. If XSLT 2.0 is the target language (which calls `fn:node-name`), the function returns an empty sequence for nodes which have no names.

Note: Getting the node name is not supported for "File input" nodes, database tables or fields, XBRL, Excel, JSON, or Protocol Buffers fields.



Languages

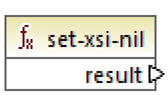
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
node	Mandatory parameter. Connect this input to the node whose name you want to get.

6.6.7.4 set-xsi-nil

Sets the target node to xsi:nil.



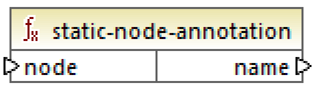
Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

6.6.7.5 static-node-annotation

Returns the string with annotation of the connected node. The input must be: (i) a source component node, or (ii) a user-defined function of type "inline[201]" that is directly connected to a parameter[204], which in turn is directly connected to a node in the calling mapping.

The connection must be direct. It cannot pass through a filter or a regular (not "inline") user-defined function. This is a pseudo-function, which is replaced at generation time with the text acquired from the connected node, and is therefore available for all languages.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

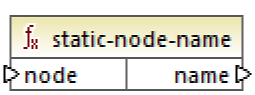
Parameters

Argument	Description
node	Mandatory parameter. Connect this input to the node whose annotation you want to get.

6.6.7.6 static-node-name

Returns the string with the name of the connected node. The input must be: (i) a source component node, or (ii) a user-defined function of type "inline[201]" that is directly connected to a parameter[204], which in turn is directly connected to a node in the calling mapping.

The connection must be direct. It cannot pass through a filter or a non-inlined user-defined function. This is a pseudo-function, which is replaced at generation time with the text acquired from the connected node, and is therefore available for all languages.



Languages

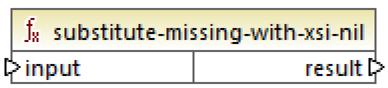
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Argument	Description
node	Mandatory parameter. Connect this input to the node whose name you want to get.

6.6.7.7 substitute-missing-with-xsi-nil

For nodes with simple content, this function substitutes any missing (or null values) of the source component, with the `xsi:nil` attribute in the target node.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

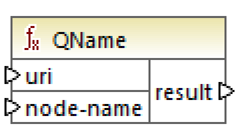
Argument	Description
input	Mandatory parameter. Connect this input to the node whose name you want to get.

6.6.8 core | QName functions

QName functions provide ways to manipulate the Qualified Names (QName) in XML documents.

6.6.8.1 QName

Constructs a QName from a namespace URI and a local part. Use this function to create a QName in a target component. The **uri** and **node-name** parameters can be supplied by a constant function.



Languages

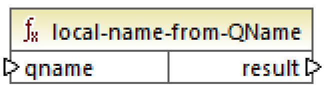
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
uri	Mandatory. Provides the URI.
node-name	Mandatory. Provides the name of the node.

6.6.8.2 local-name-from-QName

Extracts the local name part from a value of type `xs:QName`. Note that, unlike the local-name [264] function which returns the local name of the *node*, this function processes the *content* of the item connected to the **qname** input.



Languages

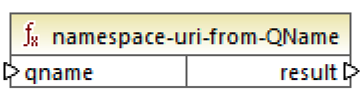
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
qname	Mandatory. Provides the function's input value, of type `xs:QName`.

6.6.8.3 namespace-uri-from-QName

Returns the namespace URI part of the QName value supplied as argument.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

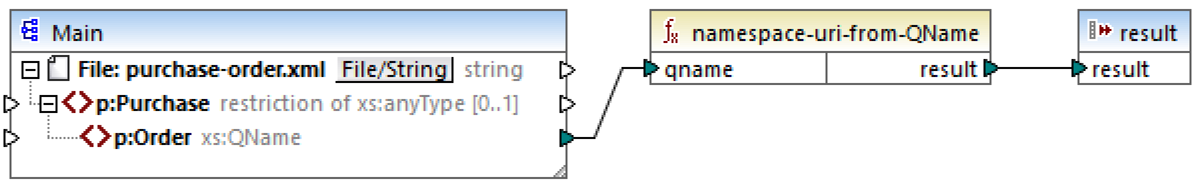
Name	Description
qname	Mandatory. Provides the function's input value.

Example

The following XML file contains a QName value, **o:name**. Note that the prefix "o" is mapped to the namespace **http://NamespaceTest.com/Order**.

```
<?xml version="1.0" encoding="utf-8"?>
<p:Purchase xsi:schemaLocation="http://NamespaceTest.com/Purchase Main.xsd"
          xmlns:p="http://NamespaceTest.com/Purchase"
          xmlns:o="http://NamespaceTest.com/Order"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
          <p:Order>o:name</p:Order>
</p:Purchase>
```

A mapping that processes the QName value and gets the namespace URI is illustrated below:



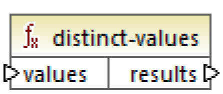
The output of this mapping is **http://NamespaceTest.com/Order**.

6.6.9 core | sequence functions

Sequence functions allow processing of input sequences [398] and grouping of their content.

6.6.9.1 distinct-values

Processes the sequence of values connected to the **values** input and returns only the distinct values, as a sequence. This is useful when you need to remove duplicate values from a sequence and copy only the unique items to the target component.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

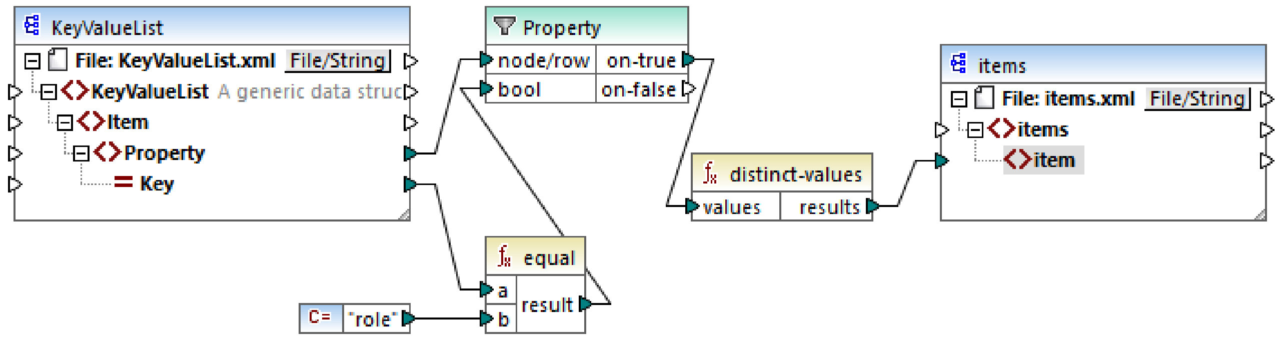
Name	Description
values	This input must receive a connection from a mapping item that provides a sequence [398] of zero or more values. For example, the connection may originate from a source XML item.

Example

The following XML file contains information about employees of a demo company. Some employees have the same role; therefore, the "role" attribute role contains duplicate values. For example, both "Loby Matise" and "Susi Sanna" have the role "Support".

```
<?xml version="1.0" encoding="UTF-8"?>
<KeyValueList xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="KeyValueList.xsd">
   <Item>
      <Property Key="role">Manager</Property>
      <Property Key="First">Vernon</Property>
      <Property Key="Last">Callaby</Property>
   </Item>
   <Item>
      <Property Key="role">Programmer</Property>
      <Property Key="First">Frank</Property>
      <Property Key="Last">Further</Property>
   </Item>
   <Item>
      <Property Key="role">Support</Property>
      <Property Key="First">Loby</Property>
      <Property Key="Last">Matise</Property>
   </Item>
   <Item>
      <Property Key="role">Support</Property>
      <Property Key="First">Susi</Property>
      <Property Key="Last">Sanna</Property>
   </Item>
</KeyValueList>
```

Let's suppose that you need to extract a list of all *unique* role names that occur in this XML file. This can be achieved with a mapping like the one below:



In the mapping above, the following happens:

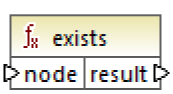
- Each **Property** element from the source XML file is processed by a filter.
- The connection to the filter's **bool** input ensures that only **Property** elements where the **Key** attribute is equal to "role" are supplied to the target component. The string "role" is provided by a constant. Note that the filter's output still produces duplicates at this stage (since there are two "Support" properties that meet the filter's condition).
- The sequence produced by the filter is processed by the **distinct-values** function, which excludes any duplicate values.

As a result, the mapping output is as follows (excluding the XML and schema declarations):

```
<items>
   <item>Manager</item>
   <item>Programmer</item>
   <item>Support</item>
</items>
```

6.6.9.2 exists

Returns **true** if the connected node exists; **false** otherwise. Since it returns a Boolean value, this function is typically used with filters [168], to filter out only records which have (or perhaps do not have) a child element or attribute.



Languages

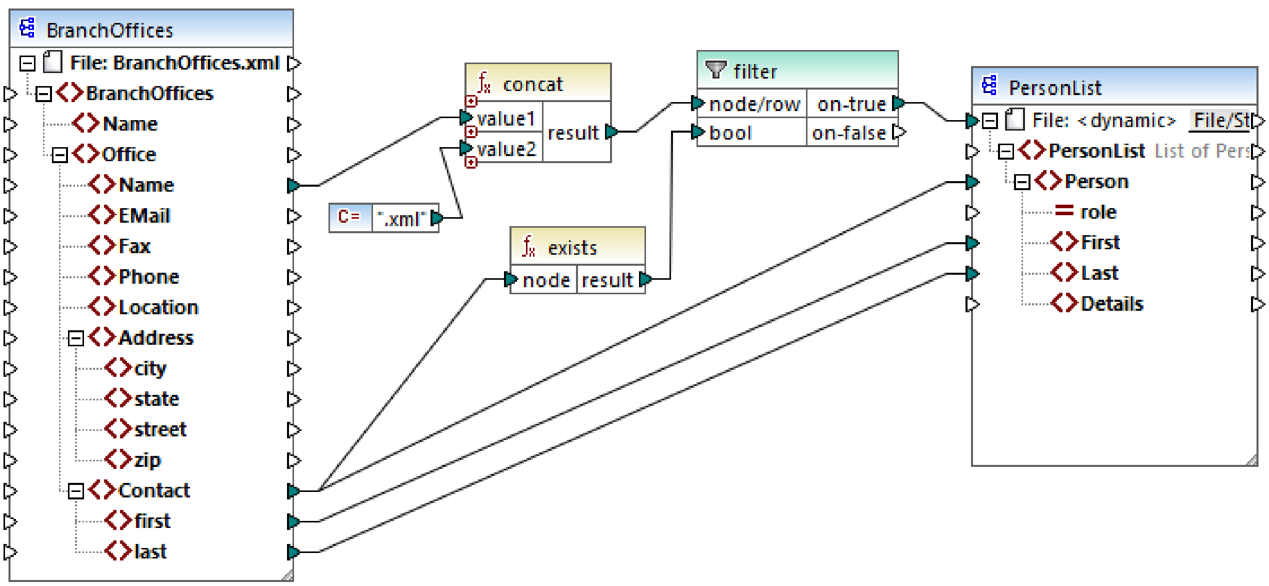
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
node	The node to be tested for existence.

Examples

The following mapping illustrates how to filter data with the help of the **exists** function. This mapping is called **PersonListsForAllBranchOffices.mfd** and it can be found in the **<Documents>\Altova\MapForce2023\MapForceExamples** directory.



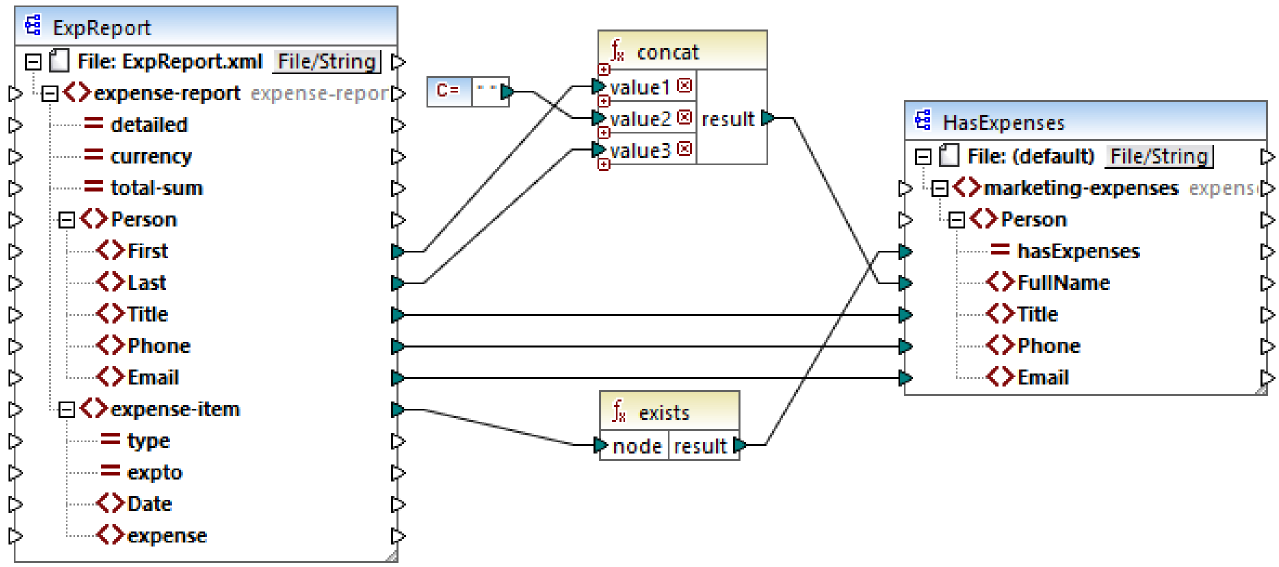
PersonListsForAllBranchOffices.mfd

In the source file **BranchOffices.xml**, there are three **Office** elements. Notably, one of the offices does not have any **Contact** child elements. The goal of the mapping is many-fold:

a) for each office, extract a list of contacts that exist in that office
b) for each office, create a separate XML file with the same name as the office
c) do not generate the XML file if the office has no contacts.

To achieve these goals, a filter was added to the mapping. The filter passes on to the target only those **Office** items where at least one **Contact** item exists. This Boolean condition is provided by the **exists** function. If the function's result is true, then the name of the office is concatenated with the string `.xml` in order to produce the target file name. For more information about generating file names from the mapping, see Processing Multiple Input or Output Files Dynamically 417 .

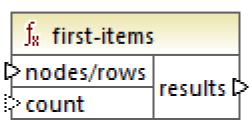
Another example is the following mapping: **<Documents>\Altova\MapForce2023\MapForceExamples\HasMarketingExpenses.mfd**. Here, if an **expense-item** exists in the source XML, then the **hasExpenses** attribute is set to **true** in the target XML file.



HasMarketingExpenses.mfd

6.6.9.3 first-items

Returns the first *N* items of the input sequence, where *N* is supplied by the **count** parameter.



Languages

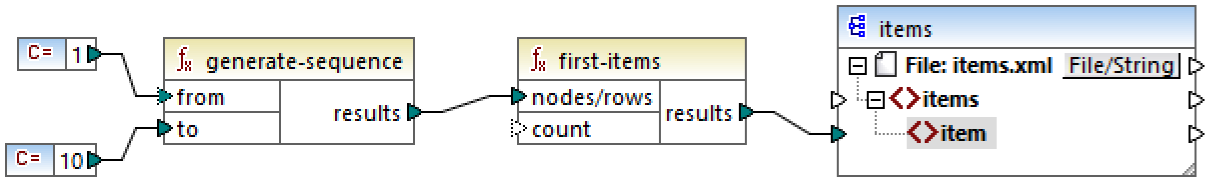
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
nodes/rows	This input must receive a connection from a mapping item that provides a sequence[398] of zero or more values. For example, the connection may originate from a source XML item.
count	Optional parameter. Specifies how many items should be retrieved from the input sequence. The default value is **1**.

Example

The following mock-up mapping generates a sequence of 10 values. The sequence is processed by the `first-items` function and the result is written to a target XML file.



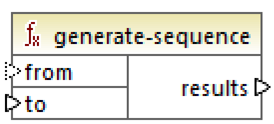
Because the **count** argument has no value, the default value of **1** applies. As a result, only the first value from the sequence is generated in the mapping output:

```
<items>
   <item>1</item>
</items>
```

For a more realistic example, see the **FindHighestTemperatures.mfd** mapping discussed in Supplying Parameters to the Mapping 139.

6.6.9.4 generate-sequence

Creates a sequence of integers using the "from" and "to" parameters as the boundaries.



Languages

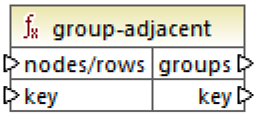
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
from	Optional parameter. Specifies the integer that the sequence should start with (lower boundary). The default value is **1**.
to	Mandatory parameter. Specifies the integer that the sequence should end with (upper boundary).

6.6.9.5 group-adjacent

The **group-adjacent** function groups the items connected to the **nodes/rows** input by the key connected to the **key** input. Note that this function places items that share the same key into separate groups if they are not adjacent. If multiple consecutive (adjacent) items share the same key, they are placed into the same group.



For example, in the abstract transformation illustrated below, the grouping key is "Department". The left side of the diagram shows the input data while the right side shows the output data after grouping. The following takes place when the transformation runs:

- Initially, the first key, "Administration", creates a new group.
- The next key is different, so a second group is created, "Marketing".
- The third key is also different, so another group is created, "Engineering".
- The fourth key is the same as the third; therefore, this record is placed in the already existing group.
- Finally, the fifth key is different from the fourth, and this creates the last group.

As illustrated below, "Michelle Butler" and "Fred Landis" were grouped together because they have the same key and are adjacent. However, "Vernon Callaby" and "Frank Further" are in separate groups because they are not adjacent, even though they have the same key.

Department	Name
Administration	Vernon Callaby
Marketing	Susi Sanna
Engineering	Michelle Butler
Engineering	Fred Landis
Administration	Frank Further

Department	Name
Administration	Vernon Callaby

Department	Name
Marketing	Susi Sanna

Department	Name
Engineering	Michelle Butler
Engineering	Fred Landis

Department	Name
Administration	Frank Further

Languages

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

Parameters

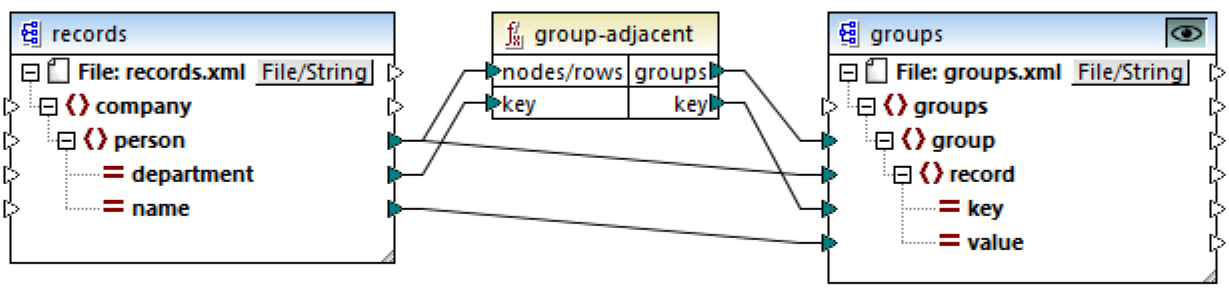
Name	Description
nodes/rows	This input must receive a connection from a mapping item that provides a sequence 398 of zero or more values. For example, the connection may originate from a source XML item.
key	The key by which to group items.

Example

Let's assume that your source data is an XML file with the following content (note that, in the code listing below, the namespace and XML declarations were removed for simplicity).

```
<company>
   <person department="Administration" name="Vernon Callaby"/>
   <person department="Marketing" name="Susi Sanna"/>
   <person department="Engineering" name="Michelle Butler"/>
   <person department="Engineering" name="Fred Landis"/>
   <person department="Administration" name="Frank Further"/>
</company>
```

The business requirement is to group person records by department, provided they are adjacent. To achieve this, the following mapping invokes the `group-adjacent` function, and supplies **department** as **key**.



The mapping result is as follows:

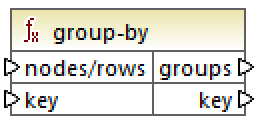
```
<groups>
   <group>
      <record key="Administration" value="Vernon Callaby"/>
   </group>
   <group>
      <record key="Marketing" value="Susi Sanna"/>
   </group>
   <group>
      <record key="Engineering" value="Michelle Butler"/>
      <record key="Engineering" value="Fred Landis"/>
   </group>
```

```
    <group>
      <record key="Administration" value="Frank Further"/>
    </group>
</groups>
```

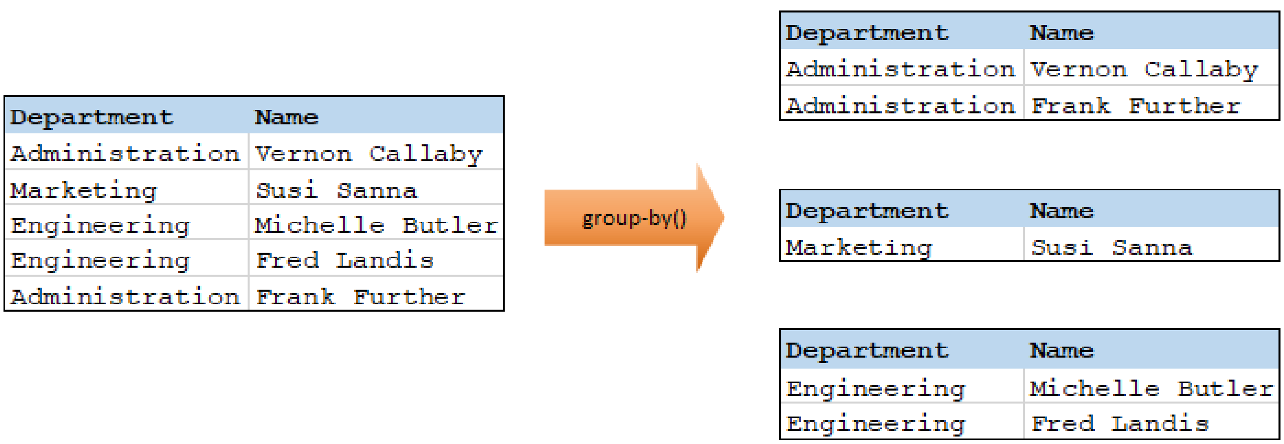
This example, together with other grouping examples, is part of the following mapping file: **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\GroupingFunctions.mfd**. Remember to click the **Preview** button applicable to the function you want to preview, before clicking the **Output** tab.

6.6.9.6 group-by

The **group-by** function creates groups of records according to some grouping key that you specify.



For example, in the abstract transformation illustrated below, the grouping key is "Department". Since there are three unique departments in total, applying the group-by function would create three groups:



Languages

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

Parameters

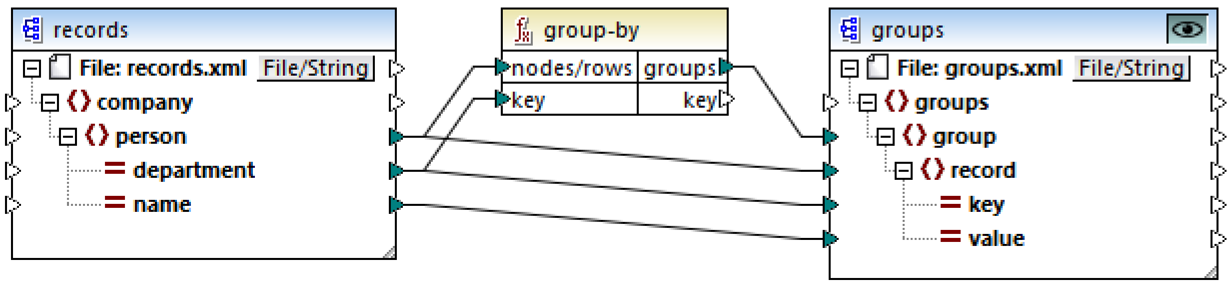
Name	Description
nodes/rows	This input must receive a connection from a mapping item that provides a sequence 398 of zero or more values. For example, the connection may originate from a source XML item.
key	The key by which to group items.

Example

Let's assume that your source data is an XML file with the following content (note that, in the code listing below, the namespace and XML declarations were removed for simplicity).

```
<company>
   <person department="Administration" name="Vernon Callaby"/>
   <person department="Marketing" name="Susi Sanna"/>
   <person department="Engineering" name="Michelle Butler"/>
   <person department="Engineering" name="Fred Landis"/>
   <person department="Administration" name="Frank Further"/>
</company>
```

The business requirement is to group person records by department. To achieve this, the following mapping invokes the **group-by** function, and supplies **department** as key.



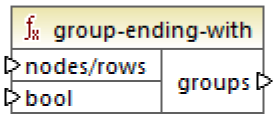
The mapping result is as follows:

```
<groups>
   <group>
      <record key="Administration" value="Vernon Callaby"/>
      <record key="Administration" value="Frank Further"/>
   </group>
   <group>
      <record key="Marketing" value="Susi Sanna"/>
   </group>
   <group>
      <record key="Engineering" value="Michelle Butler"/>
      <record key="Engineering" value="Fred Landis"/>
   </group>
</groups>
```

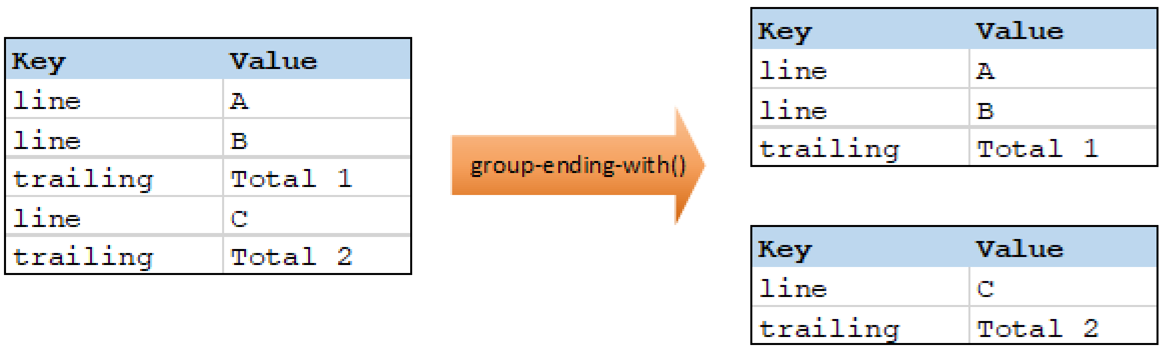
This example, together with other grouping examples, is part of the following mapping file: **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\GroupingFunctions.mfd**. Remember to click the **Preview**  button applicable to the function you want to preview, before clicking the **Output** tab.

6.6.9.7 group-ending-with

The **group-ending-with** function takes a Boolean condition as argument. If the Boolean condition is true, a new group is created, ending with the record that satisfies the condition.



In the example below, the condition is that "Key" must be equal to "trailing". This condition is true for the third and fifth records, so two groups are created as a result:



Note: One additional group is created if records exist after the last one that satisfies the condition. For example, if there were more "line" records after the last "trailing" record, these would all be placed into a new group.

Languages

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
nodes/rows	This input must receive a connection from a mapping item that provides a sequence [398] of zero or more values. For example, the connection may originate from a source XML item.
bool	Provides the Boolean condition that starts a new group when **true**.

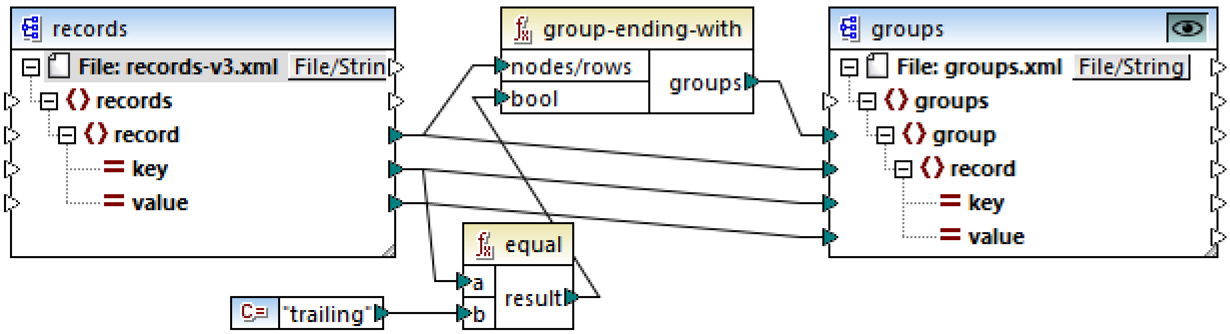
Example

Let's assume that your source data is an XML file with the following content (note that, in the code listing below, the namespace and XML declarations were removed for simplicity).

```
<records>
   <record key="line" value="A"/>
```

```
    <record key="line" value="B"/>
    <record key="trailing" value="Total 1"/>
    <record key="line" value="C"/>
    <record key="trailing" value="Total 2"/>
</records>
```

The business requirement is to create groups for each "trailing" record. Each group must also include any "line" records that precede the "trailing" record. To achieve this, the following mapping invokes the `group-ending-with` function. In the mapping below, whenever the **key** name is equal to "trailing", the argument supplied to **bool** becomes **true**, and a new group is created.



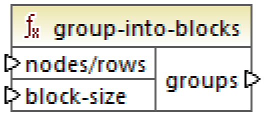
The mapping result is as follows:

```
<groups>
    <group>
        <record key="line" value="A"/>
        <record key="line" value="B"/>
        <record key="trailing" value="Total 1"/>
    </group>
    <group>
        <record key="line" value="C"/>
        <record key="trailing" value="Total 2"/>
    </group>
</groups>
```

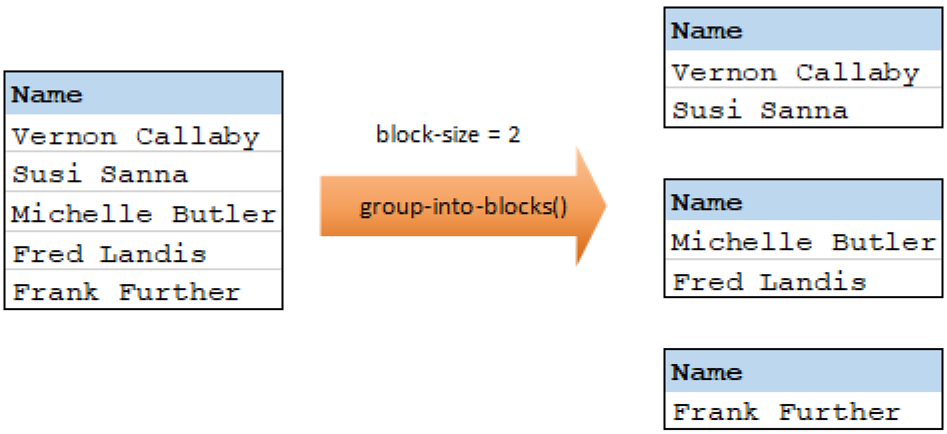
This example, together with other grouping examples, is part of the following mapping file: **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\GroupingFunctions.mfd**. Remember to click the **Preview**  button applicable to the function you want to preview, before clicking the **Output** tab.

6.6.9.8 group-into-blocks

The `group-into-blocks` function creates equal groups that contain exactly N items, where N is the value you supply to the `block-size` argument. Note that the last group may contain N items or less, depending on the number of items in the source.



In the example below, `block-size` is 2. Since there are five items in total, each group contains exactly two items, except for the last one.



Languages

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

Parameters

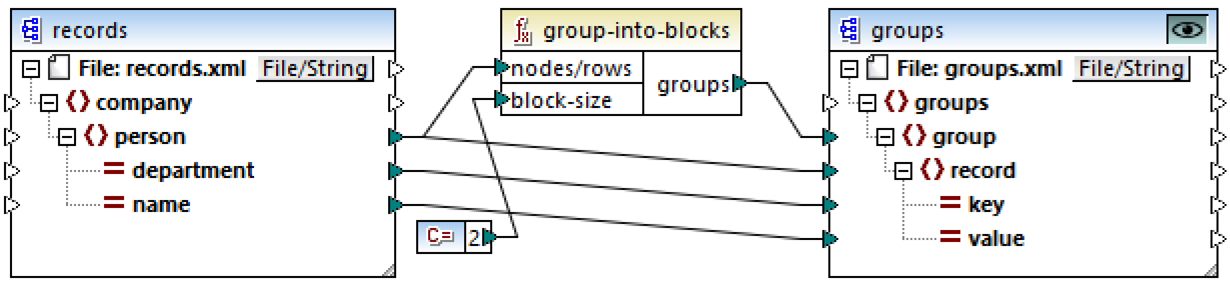
Name	Description
nodes/rows	This input must receive a connection from a mapping item that provides a sequence [398] of zero or more values. For example, the connection may originate from a source XML item.
block-size	Specifies the size of each group

Example

Let's assume that your source data is an XML file with the following content (note that, in the code listing below, the namespace and XML declarations were removed for simplicity).

```
<company>
   <person department="Administration" name="Vernon Callaby"/>
   <person department="Marketing" name="Susi Sanna"/>
   <person department="Engineering" name="Michelle Butler"/>
   <person department="Engineering" name="Fred Landis"/>
   <person department="Administration" name="Frank Further"/>
</company>
```

The business requirement is to group person records into blocks of two items each. To achieve this, the following mapping invokes the **group-into-blocks** function, and supplies the integer value "2" as **block-size**.



The mapping result is as follows:

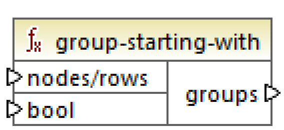
```
<groups>
   <group>
      <record key="Administration" value="Vernon Callaby"/>
      <record key="Marketing" value="Susi Sanna"/>
   </group>
   <group>
      <record key="Engineering" value="Michelle Butler"/>
      <record key="Engineering" value="Fred Landis"/>
   </group>
   <group>
      <record key="Administration" value="Frank Further"/>
   </group>
</groups>
```

Note that the last group contains only one item, since the total number of items (5) cannot be divided evenly by 2.

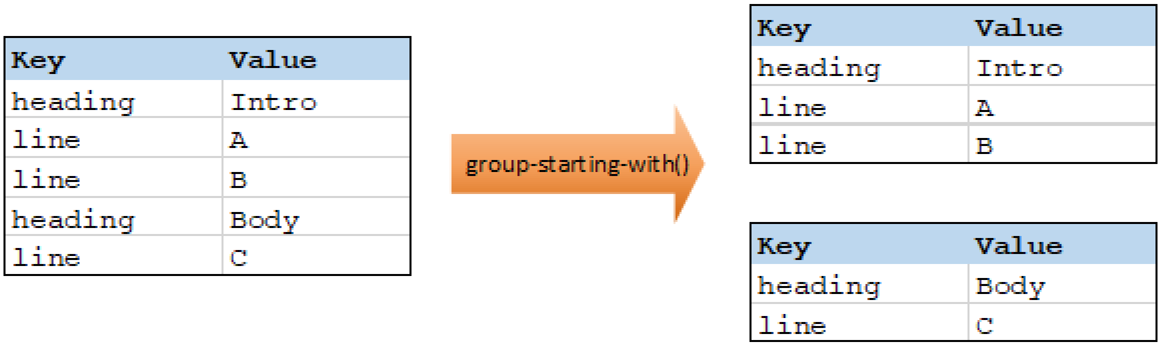
This example, together with other grouping examples, is part of the following mapping file: **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\GroupingFunctions.mfd**. Remember to click the **Preview** button applicable to the function you want to preview, before clicking the **Output** tab.

6.6.9.9 group-starting-with

The **group-starting-with** function takes a Boolean condition as argument. If the Boolean condition is true, a new group is created, starting with the record that satisfies the condition.



In the example below, the condition is that "Key" must be equal to "heading". This condition is true for the first and fourth records, so two groups are created as a result:



Note: One additional group is created if records exist before the first one that satisfies the condition. For example, if there were more "line" records before the first "heading" record, these would all be placed into a new group.

Languages

Built-in, C++, C#, Java, XSLT 2.0, XSLT 3.0.

Parameters

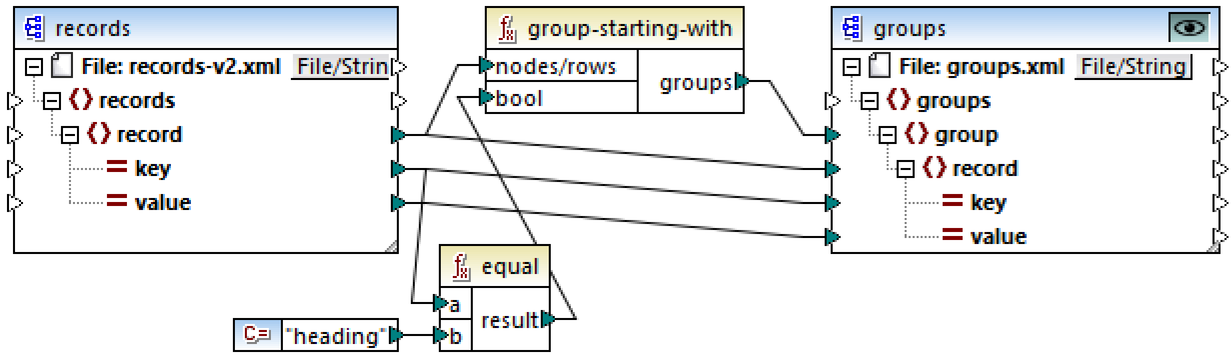
Name	Description
nodes/rows	This input must receive a connection from a mapping item that provides a sequence [398] of zero or more values. For example, the connection may originate from a source XML item.
bool	Provides the Boolean condition that starts a new group when **true**.

Example

Let's assume that your source data is an XML file with the following content (note that, in the code listing below, the namespace and XML declarations were removed for simplicity).

```
<records>
   <record key="heading" value="Intro"/>
   <record key="line" value="A"/>
   <record key="line" value="B"/>
   <record key="heading" value="Body"/>
   <record key="line" value="C"/>
</records>
```

The business requirement is to create groups for each "heading" record. Each group must also include any "line" records that follow the "heading" record. To achieve this, the following mapping invokes the `group-starting-with` function. In the mapping below, whenever the **key** name is equal to "heading", the argument supplied to **bool** becomes **true**, and a new group is created.



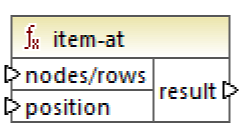
The mapping result is as follows:

```
<groups>
   <group>
      <record key="heading" value="Intro"/>
      <record key="line" value="A"/>
      <record key="line" value="B"/>
   </group>
   <group>
      <record key="heading" value="Body"/>
      <record key="line" value="C"/>
   </group>
</groups>
```

This example, together with other grouping examples, is part of the following mapping file: **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\GroupingFunctions.mfd**. Remember to click the **Preview**  button applicable to the function you want to preview, before clicking the **Output** tab.

6.6.9.10 item-at

Returns an item from the sequence of **nodes/rows** supplied as argument, at the position supplied by the **position** argument. The first item is at position **1**.



Languages

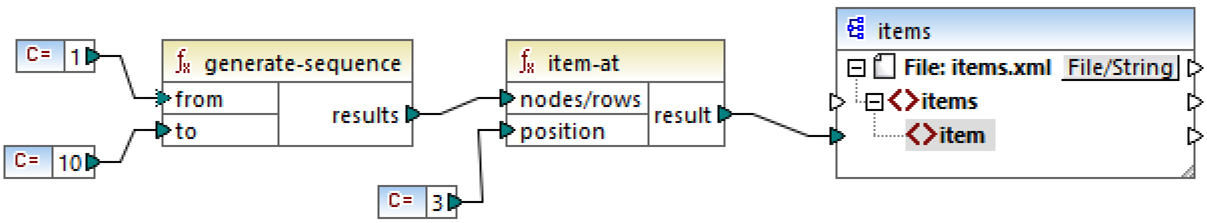
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
nodes/rows	This input must receive a connection from a mapping item that provides a sequence[398] of zero or more values. For example, the connection may originate from a source XML item.
position	This integer specifies which item from the sequence of items is to be returned.

Example

The following mock-up mapping generates a sequence of 10 values. The sequence is processed by the `item-at` function and the result is written to a target XML file.

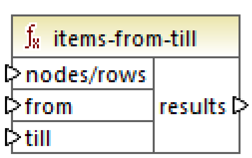


Because the **position** argument is set to **3**, only the third value from the sequence is passed on to the target. Consequently, the mapping output is as follows (excluding the XML and schema declarations):

```
<items>
   <item>3</item>
</items>
```

6.6.9.11 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**.



Languages

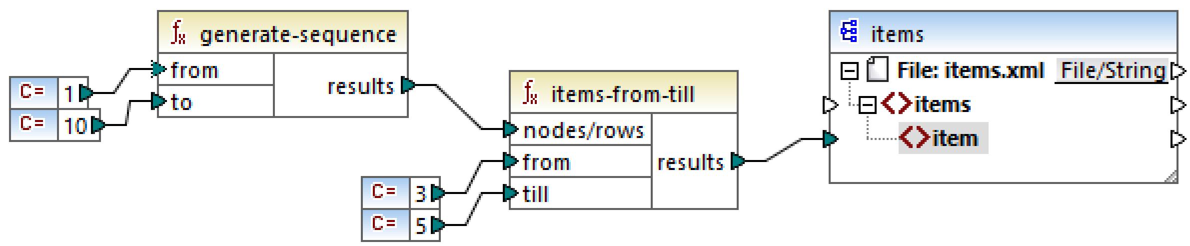
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
nodes/rows	This input must receive a connection from a mapping item that provides a sequence 398 of zero or more values. For example, the connection may originate from a source XML item.
from	This integer specifies the starting position from which items must be retrieved.
till	This integer specifies the position up to which items must be retrieved.

Example

The following mock-up mapping generates a sequence of 10 values. The sequence is processed by the **`items-from-till`** function and the result is written to a target XML file.

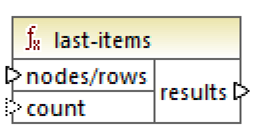


Because the **from** and **till** arguments are set to **3** and **5**, respectively, only the subset of values from **3** through **5** are passed on to the target. Consequently, the mapping output is as follows (excluding the XML and schema declarations):

```
<items>
   <item>3</item>
   <item>4</item>
   <item>5</item>
</items>
```

6.6.9.12 last-items

Returns the last *N* items of the input sequence, where *N* is supplied by the **count** parameter. The first item is at position "1".



Languages

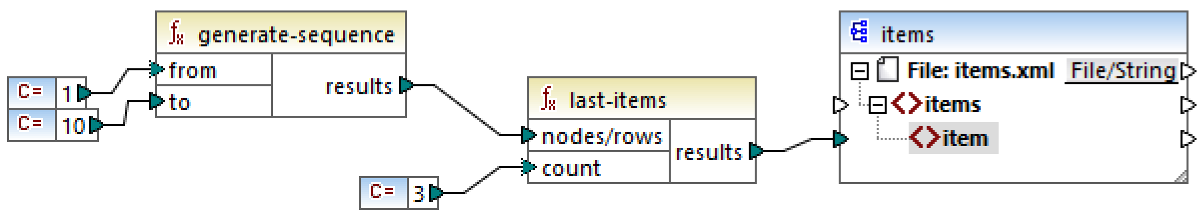
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
nodes/rows	This input must receive a connection from a mapping item that provides a sequence[398] of zero or more values. For example, the connection may originate from a source XML item.
count	Optional parameter. Specifies how many items should be retrieved from the input sequence. The default value is **1**.

Example

The following mock-up mapping generates a sequence of 10 values. The sequence is processed by the **`last-items`** function and the result is written to a target XML file.

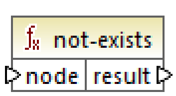


Because the count argument is set to **3**, only the last three values from the sequence are passed on to the target. Consequently, the mapping output is as follows (excluding the XML and schema declarations):

```
<items>
   <item>8</item>
   <item>9</item>
   <item>10</item>
</items>
```

6.6.9.13 not-exists

Returns **false** if the connected node exists; **true** otherwise. This function is the opposite of exists[271] function, but, otherwise, it has the same use.



Languages

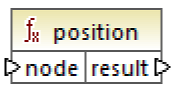
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
node	The node to be tested for non-existence.

6.6.9.14 position

Returns the position of an item within the sequence of items currently being processed. This can be used, for example, to auto-number items sequentially.



Languages

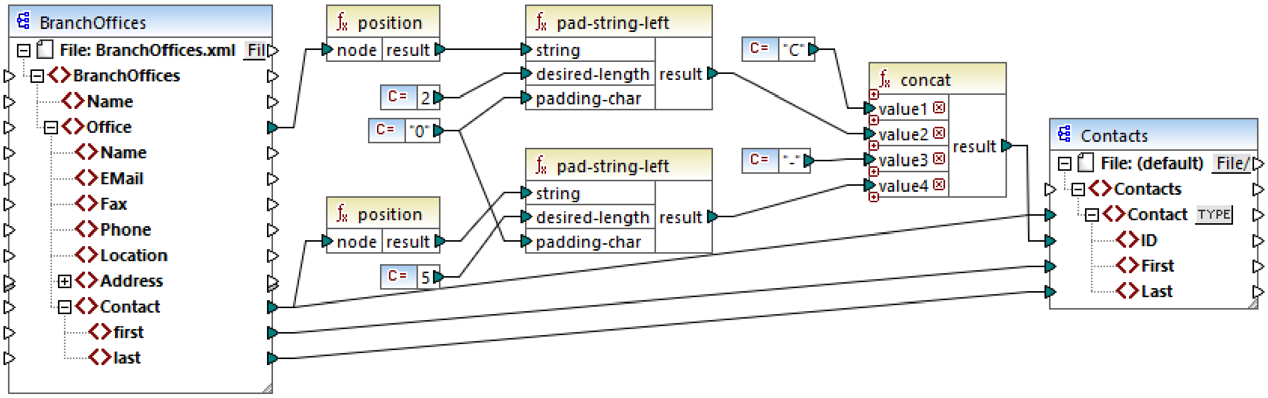
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
node	This input must receive a connection from a mapping item that provides a sequence [398] of zero or more values. For example, the connection may originate from a source XML item.

Example

The following mapping illustrates using the `position` function in order to generate unique identification values in data generated by the mapping. This mapping is accompanied by a mapping design file that is available at the following path:
<Documents>\Altova\MapForce2023\MapForceExamples\ContactsFromBranchOffices.mfd.



ContactsFromBranchOffices.mfd

In the mapping above, the source XML file contains three branch offices. A branch office may contain an arbitrary number of **Contact** child items. The goals of the mapping are as follows:

- Extract all **Contact** items from the source XML file and write them to the target XML file.
- Each contact must be assigned a unique identification number (the **ID** item in the target XML).
- The ID of each contact must take the form **`CXX-YYYYY`**, where X identifies the office number, and Y identifies the contact number. If the office number is less than two characters, it must be left-padded with zeros. Likewise, if the contact number takes less than five characters, it must be left-padded with zeros. Consequently, a valid identification number of the first contact from the first office should look like **`C01-00001`**.

To achieve the mapping goals, several MapForce functions have been used, including the **`position`** function. The upper **`position`** function gets the position of each office. The lower one gets the position of each contact, in the context of each office.

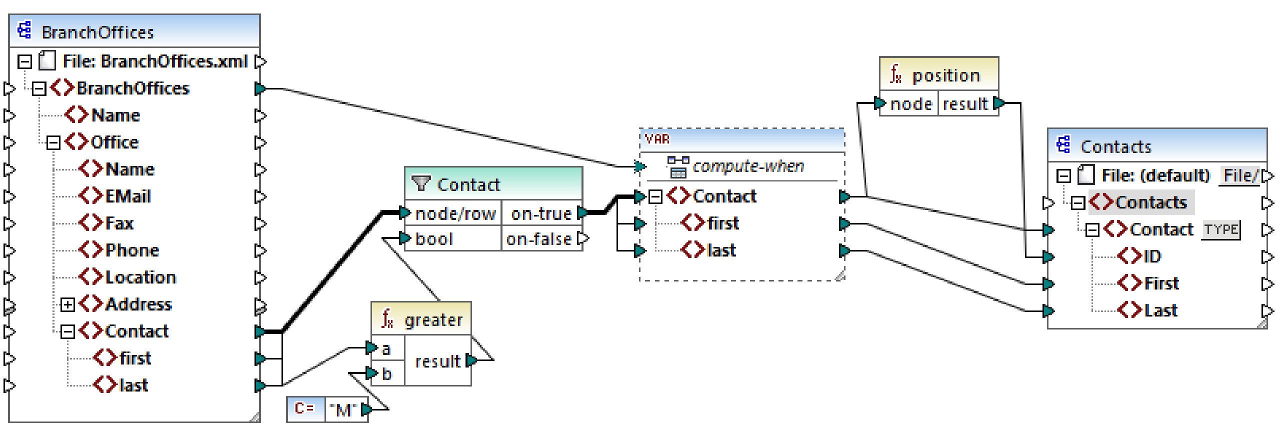
When using the **`position`** function, it is important to consider the current mapping context[399]. More specifically, when the mapping runs, the initial mapping context is established from the root item of the target component to the source item connected to it (even indirectly via functions). In this example, the upper **`position`** function processes *the sequence of all offices* and it initially generates the value 1, corresponding to the first office in the sequence. The lower **`position`** function generates sequential numbers corresponding to the contact's position *in the context of that office* (1, 2, 3, and so on). Note that this "inner" sequence will be reset (and thus start from 1 again) when the next office gets processed. Both **`pad-string-left`** functions apply padding to the generated numbers, according to the requirements stated previously. The `concat` function operates *in the context of each contact* (because of the parent connection from the source to the target **Contact**). It joins all the computed values and returns the unique identification number of each contact.

The output generated from the mapping above is shown below (note that some of the records were removed for readability):

```
<Contacts>
   <Contact>
      <ID>C01-00001</ID>
      <First>Vernon</First>
      <Last>Callaby</Last>
   </Contact>
   <Contact>
```

```
        <ID>C01-00002</ID>
        <First>Frank</First>
        <Last>Further</Last>
    </Contact>
    <!-- ... -->
    <Contact>
        <ID>C02-00001</ID>
        <First>Steve</First>
        <Last>Meier</Last>
    </Contact>
    <Contact>
        <ID>C02-00002</ID>
        <First>Theo</First>
        <Last>Bone</Last>
    </Contact>
    <!-- ... -->
</Contacts>
```

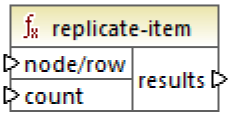
There may also be cases where you need to get the position of items resulting after applying a [filter](168)[168]. Note that the filter component is not a sequence function, and it cannot be used *directly* in conjunction with the **`position`** function to find the position of filtered items. Indirectly, this is possible by adding a variable[150] component to the mapping. For example, the mapping below is a simplified version of the previous one. Its mapping design file is available at the following path: **<Documents>\Altova\MapForce2023\MapForceExamples\PositionInFilteredSequence.mfd**.



The result of variables in MapForce are always sequences. Therefore, in the mapping above, the **`position`** function iterates through the sequence created by the variable and returns the position of each item in that sequence. This mapping is discussed in more detail in Example: Filtering and Numbering Nodes[158].

6.6.9.15 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** input, 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** input, 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`. It is also possible to supply a different **count** value for each item in the input sequence, as illustrated in the example below.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

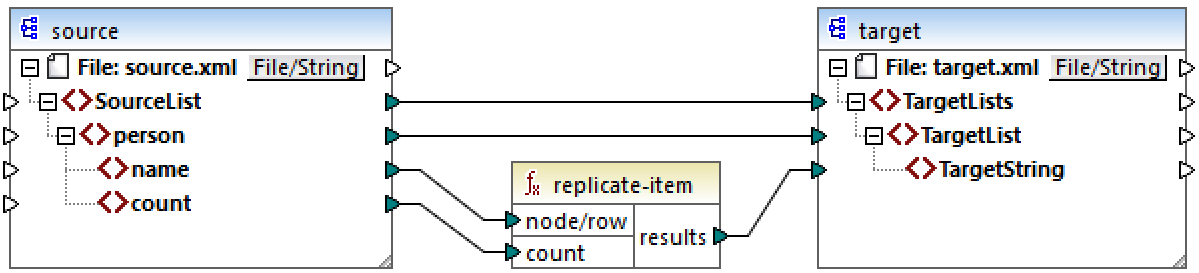
Name	Description
node/row	This input must receive a connection from a mapping item that provides a sequence [398] of zero or more values. For example, the connection may originate from a source XML item.
count	Specifies the number of times to replicate each item or sequence connected to **node/row**.

Example

Let's assume that you have a source XML file with the following structure:

```
<SourceList>
   <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:

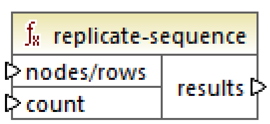


The output is as follows:

```
<TargetLists>
   <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>
```

6.6.9.16 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`.



Languages

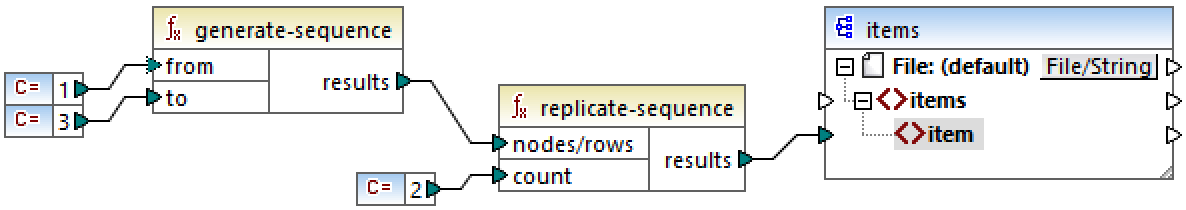
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
node/rows	This input must receive a connection from a mapping item that provides a sequence 398 of zero or more values. For example, the connection may originate from a source XML item.
count	Specifies the number of times to replicate the connected sequence.

Example

The following mock-up mapping generates the sequence `1,2,3`. The sequence is processed by the `replicate-sequence` function and the result is written to a target XML file.

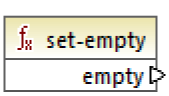


Because the **count** argument is set to **2**, the sequence is replicated twice and then passed on to the target. Consequently, the mapping output is as follows (excluding the XML and schema declarations):

```
<items>
   <item>1</item>
   <item>2</item>
   <item>3</item>
   <item>1</item>
   <item>2</item>
   <item>3</item>
</items>
```

6.6.9.17 set-empty

Returns an empty sequence.

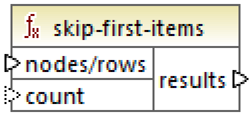


Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

6.6.9.18 skip-first-items

Skips the first N items of the input sequence, where N is supplied by the **count** argument, and returns the rest of the sequence.



Languages

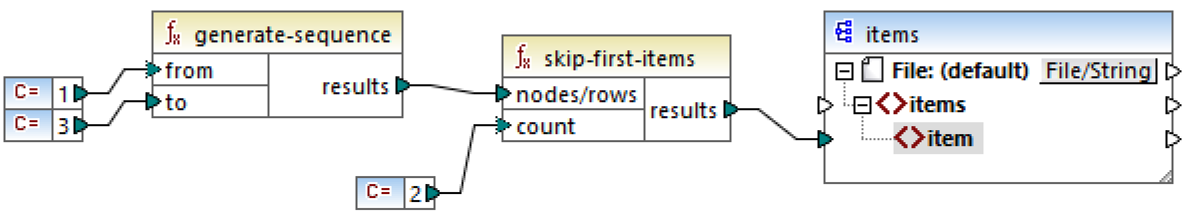
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
node/rows	This input must receive a connection from a mapping item that provides a sequence[398] of zero or more values. For example, the connection may originate from a source XML item.
count	Optional argument. Specifies the number of items to skip. The default value is **1**.

Example

The following mock-up mapping generates the sequence `1,2,3`. The sequence is processed by the `skip-first-items` function and the result is written to a target XML file.

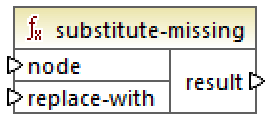


Because the **count** argument is set to **2**, the first two items are skipped and the remaining items are passed on to the target. Consequently, the mapping output is as follows (excluding the XML and schema declarations):

```
<items>
   <item>3</item>
</items>
```

6.6.9.19 substitute-missing

This function is a convenient combination of exists[271] function and if-else condition[171]. If the item connected to the **node** input exists, its content will be copied to the target. Otherwise, the content of the item connected to the **replace-with** input will be copied to the target.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

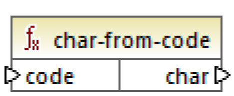
Name	Description
node	This input must receive a connection from a mapping item that provides a sequence[398] of zero or more values. For example, the connection may originate from a source XML item.
replace-with	This input must receive a connection from a mapping item that provides the replacement value.

6.6.10 core | string functions

The string functions allow you to manipulate string data so as to extract parts of strings, test for sub-strings, retrieve information from strings, split strings, and others.

6.6.10.1 char-from-code

Returns the character representation of the decimal Unicode value (code) supplied as argument. **Tip:** To find the Unicode decimal code of a character, you can use the `code-from-char`[296] function.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

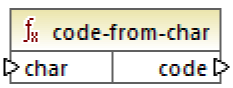
Name	Description
code	The Unicode value, as a decimal number.

Example

According to the charts available on the Unicode website (https://www.unicode.org/charts/), the exclamation mark character has the hexadecimal value of **`0021`**. The corresponding value in decimal format is **33**. Therefore, supplying **`33`** as argument to the **`char-from-code`** function will return the **`!`** character.

6.6.10.2 code-from-char

Returns the decimal Unicode value (code) of the character supplied as argument. If the string supplied as argument has multiple characters, then the code of the first character is returned.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

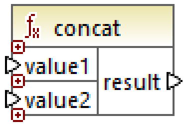
Name	Description
char	The input string value.

Example

If the input **char** is the **`$`** (dollar sign) character, the function returns **36** (which is the decimal Unicode value for this character).

6.6.10.3 concat

Concatenates (appends) two or more values into a single result string. All input values are automatically converted to type "string". By default, this function has only two parameters, but you can add more. Click **Add parameter** ( ) or **Delete parameter** ( ) to add or remove parameters.



Note: All the inputs to the `concat` function must have a value. If any of the inputs does not have a value, the function is not called and an error occurs. Be aware that an empty string is a valid input value; however, an empty sequence (such as the result of the **`set-empty`** function) is not a valid value and the function will fail as a result. To prevent this from happening, you can first process values with the substitute-missing 295 function and then supply the result as input to the **`concat`** function.

Languages

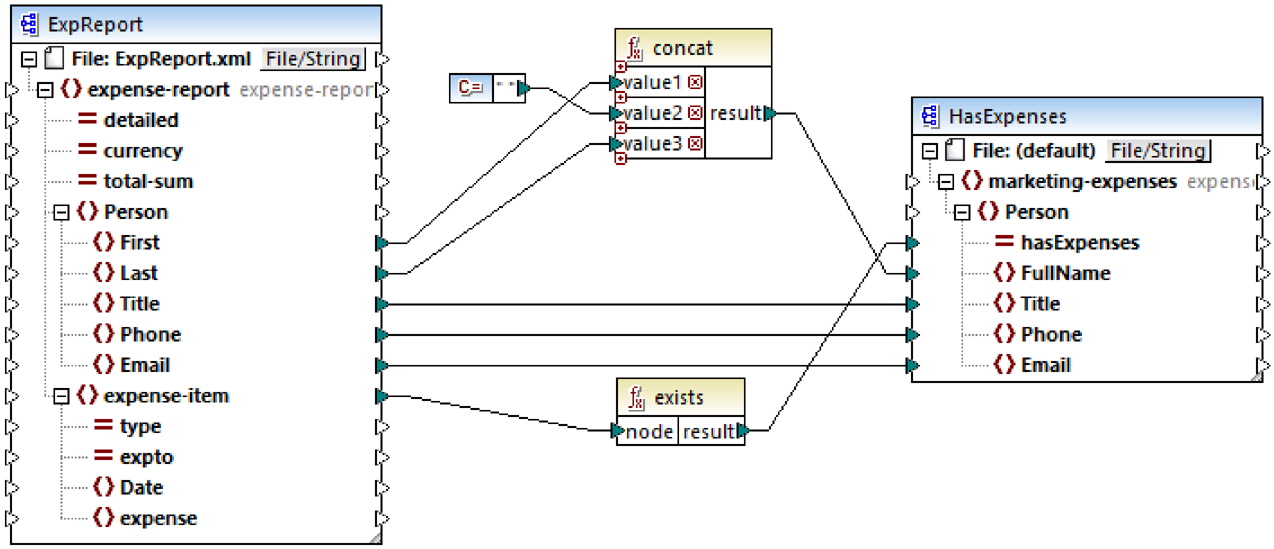
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
value1	The first input value.
value2	The second input value.
valueN	The *N* input value.

Example

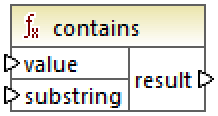
In the mapping illustrated below, the **`concat`** function joins the first name, the constant " ", and the last name. The returning value is then written to the **FullName** target item. The mapping of this function is available at the following path: **<Documents>\Altova\MapForce2023\MapForceExamples\HasMarketingExpenses.mfd**.



HasMarketingExpenses.mfd

6.6.10.4 contains

Returns Boolean **true** if the string value supplied as argument contains the sub-string supplied as argument.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

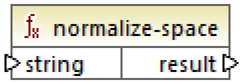
Name	Description
value	The input value (that is, the "haystack").
substring	The sub-string to look for (that is, the "needle").

Example

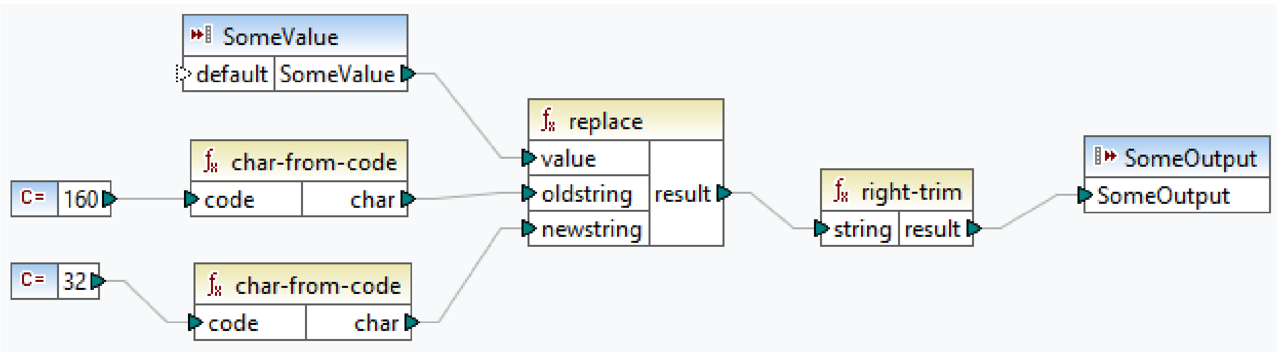
If the input **value** is "category" and **substring** is "cat", the function returns **true**.

6.6.10.5 normalize-space

The **normalize-space** function (*see screenshot below*) removes leading and trailing spaces of a string and replaces internal whitespaces with a single whitespace character. Whitespace includes space (U+0020), tab (U+0009), carriage return (U+000D), and line feed (U+000A) characters. For details about whitespaces, see the XML Recommendation.



About non-breaking spaces
The **left-trim**, **right-trim**, and **normalize-space** functions do not remove non-breaking spaces. One of the possible solutions could be to replace the non-breaking space character, whose decimal representation is 160, with the space character, whose decimal representation is 32. The mapping below shows that after the non-breaking space has been replaced, the trimmed `SomeValue` value will be mapped to the target.



If your source component is an Excel file, you can remove extra spaces in Excel using a combination of TRIM, CLEAN, and SUBSTITUTE functions. For details, see *Removing Spaces and Nonprinting Characters from Text*.

Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

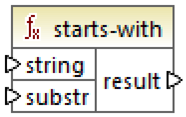
Name	Description
string	The input string to normalize.

Example

If the input string is "`  The quick   brown fox  `", the function returns "`The quick brown fox`".

6.6.10.6 starts-with

Returns Boolean **true** if the string supplied as argument starts with the sub-string supplied as argument; **false** otherwise.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

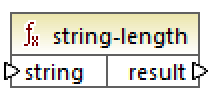
Name	Description
string	The input string.
substr	The sub-string to check for.

Example

If the input **string** is **`category`** and **substr** is **`cat`**, the function returns **true**.

6.6.10.7 string-length

Returns the number of characters in the string supplied as argument.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

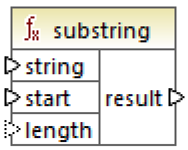
Name	Description
string	The input string.

Example

If the input string is **`car`**, the function returns **3**. If the input string is an empty string, the function returns **0**.

6.6.10.8 substring

Returns the portion of the string specified by the **start** and **length** parameters.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

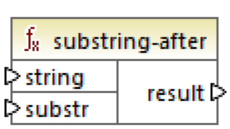
Name	Description
string	The input string.
start	Specifies the starting position (index) from which the sub-string should be retrieved. The first index is **1**.
length	Optional. Specifies the number of characters to retrieve. If the **length** parameter is not specified, the result is a fragment starting from **start** until the end of the string.

Example

If the input string is **`MapForce`**, start is **1**, and length is **3**, the function returns **`Map`**. If the input string is **`MapForce`**, start is **4**, and length is not provided, the function returns **`Force`**.

6.6.10.9 substring-after

Returns the portion of the string that occurs after the first occurrence of **substr**. If **substr** does not occur in **string**, the function returns an empty string.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

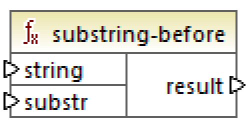
Name	Description
string	The input string.
substr	The sub-string. Any characters after the first occurrence of **substr** are the result of the function.

Example

If the input string is `MapForce`, and **substr** is `Map`, the function returns `Force`. If the input string is `2020/01/04` and **substr** is `/`, the function returns `01/04`.

6.6.10.10 substring-before

Returns the portion of the string that occurs before the first occurrence of **substr**. If **substr** does not occur in **string**, the function returns an empty string.



Languages

Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

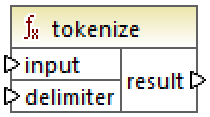
Name	Description
string	The input string.
substr	The sub-string. Any characters before the first occurrence of **substr** are the result of the function.

Example

If the input string is `MapForce`, and **substr** is `Force`, the function returns `Map`. If the input string is `2020/01/04` and **substr** is `/`, the function returns `2020`.

6.6.10.11 tokenize

Splits the input string into a sequence of strings using the delimiter supplied as argument.



Languages

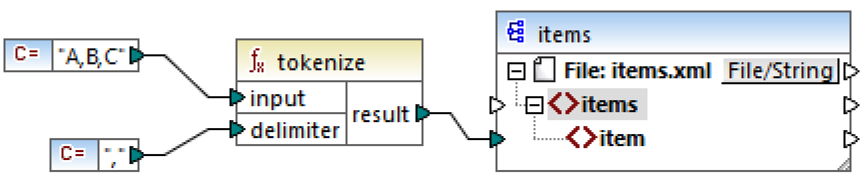
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
input	The input string.
delimiter	The delimiter to use.

Example

If the input string is `A,B,C` and the delimiter is `,` then the function returns a sequence of three strings: `A`, `B`, and `C`.

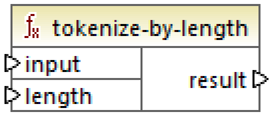


In the mock-up mapping illustrated above, the function's result is a sequence of strings. According to the general mapping rules[398], for each item in the source sequence, a new **item** is created in the target component. Consequently, the mapping output looks as follows:

```
<items>
   <item>A</item>
   <item>B</item>
   <item>C</item>
</items>
```

6.6.10.12 tokenize-by-length

Splits the input string into a sequence of strings. The size of each resulting string is determined by the **length** parameter.



Languages

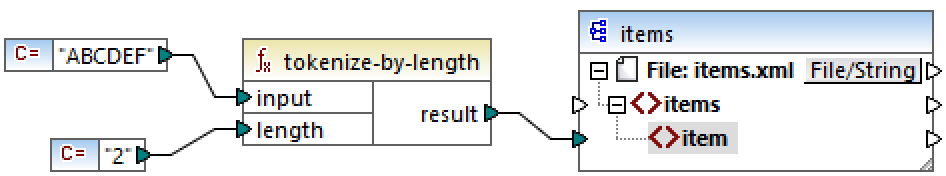
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
input	The input string.
length	Determines the length of each string in the generated sequence of strings.

Example

If the input string is **ABCDEF** and the length is **2**, then the function returns a sequence of three strings: **AB**, **CD**, and **EF**.



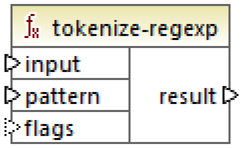
In the mock-up mapping illustrated above, the function's result is a sequence of strings. According to the general mapping rules398, for each item in the source sequence, a new **item** is created in the target component. Consequently, the mapping output looks as follows:

```
<items>
   <item>AB</item>
   <item>CD</item>
   <item>EF</item>
</items>
```

6.6.10.13 tokenize-regexp

Splits the input string into a sequence of strings. Any substring that matches the regular expression **pattern** supplied as argument defines the separator. The matched (separator) strings are not included in the result returned by the function.

Note: When generating C++, C#, or Java code, the advanced features of the regular expression syntax might differ slightly. See the regex documentation of each language for more information.



Languages

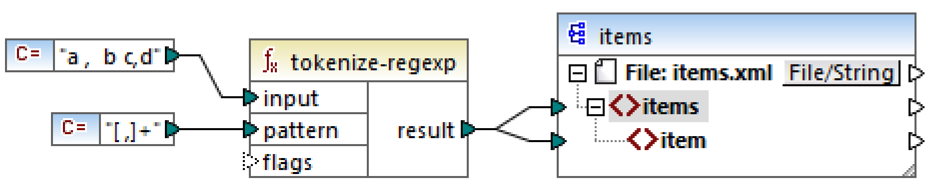
Built-in, C++, C#, Java, XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
input	The input string.
pattern	Provides a regular expression pattern. Any substring that matches the pattern will be treated as delimiter. For more information, see Regular expressions [221].
flags	Optional parameter. Provides the regular expression flags [223] to be used. For example, the flag "i" instructs the mapping process to operate in case-insensitive mode.

Example

The goal of the mapping illustrated below is to split the string `a ,  b c,d` into a sequence of strings, where each alphabetic character is an item in the sequence. Any redundant whitespace or commas must be removed.



To achieve this goal, the regular expression pattern `[ ,]+` was supplied as parameter to the **tokenize-regexp** function. This pattern has the following meaning:

- It matches any of the characters inside the character class `[ ,]`. Therefore, a split will occur whenever a comma or a space is encountered in the input string.
- The quantifier `+` specifies that one or more occurrences of the preceding character class are to be matched. Without this quantifier, each occurrence of space or comma would create a separate item in the resulting sequence of strings, which is not the intended result.

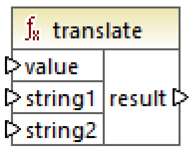
The mapping output is as follows:

```
<items>
   <item>a</item>
```

```
    <item>b</item>
    <item>c</item>
    <item>d</item>
</items>
```

6.6.10.14 translate

Performs a character by character replacement. It looks in the **value** for characters contained in **string1**, and replaces each character with the one in the same position in the **string2**. When there are no corresponding characters in **string2**, the character is removed.



Languages

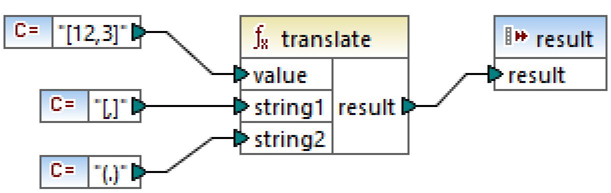
Built-in, C++, C#, Java, XQuery, XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Name	Description
value	The input string.
string1	Provides a list of search characters. The position of each character inside the string is important.
string2	Provides a list of replacement characters. The position of each replacement character must correspond to the one in **string1**.

Example

Let's suppose you want to convert the string `[12,3]` to `(12.3)`. Namely, the square brackets must be replaced by round brackets, and any comma must be replaced by the dot character. To achieve this, you can call the `translate` function as follows:



In the mapping above, the first constant supplies the input string to be processed. The second and the third constant provide a list of characters as **string1** and **string2**, respectively.

string1	`[,]`
string2	`(.)`

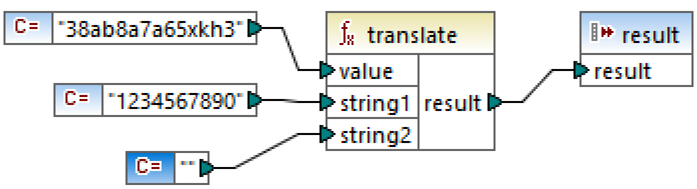
Notice that both **string1** and **string2** have the same number of characters. For each character in **string1**, the equivalent character at the same position from **string2** will be used as a replacement. Consequently, the following replacements will take place:

- Each `[` will be replaced by a `(`
- Each `,` will be replaced by a `.`
- Each `]` will be replaced by a `)`

The mapping output is as follows:

```
(12.3)
```

This function can also be used to strip certain characters selectively from a string. To achieve this, set the **string1** parameter to the characters you want to remove, and **string2** to an empty string. For example, the mapping below removes all digits from the string **`38ab8a7a65xkh3`**.



The mapping output is as follows:

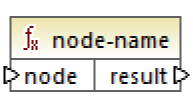
```
abaaxkh
```

6.6.11 xpath2 | accessors

Functions from the **xpath2 | accessors** sub-library retrieve information about XML nodes or items. These functions are available when either the XSLT2 or XQuery languages are selected.

6.6.11.1 base-uri

The **`base-uri`** function takes a node as input, and returns the URI of the XML resource containing the node. The output is of type `xs:string`.



Languages

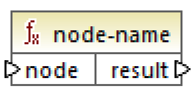
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
node	`mf:node`	The input node.

6.6.11.2 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`**[268] function.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

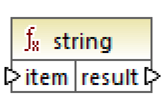
Parameters

Name	Type	Description
node	`mf:node`	The input node.

6.6.11.3 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.)



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
item	`mf:item`	The input value.

6.6.12 xpath2 | anyURI functions

The **xpath2 | anyURI** sub-library contains the **`resolve-uri`** function. This function is available when either the XSLT2 or XQuery languages are selected.

6.6.12.1 resolve-uri

The **`resolve-uri`** function takes a relative URI as its first argument and resolves it against the base URI in the second argument. The result is of data type `xs:string`. The function's implementation treats both inputs as strings; no checks are performed as to whether the resources identified by these URIs actually exist.

Languages

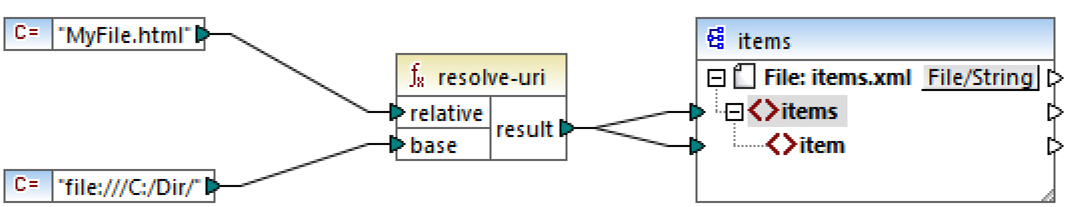
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
relative	`xs:string`	The relative URI to be resolved against the base.
base	`xs:string`	The base URI.

Example

In the mapping illustrated below, the first argument provides the relative URI **`MyFile.html`**, and the second argument provides the base URI **`file:///C:/Dir/`**. The resolved URI will be a concatenation of both, so **`file:///C:/Dir/MyFile.html`**.

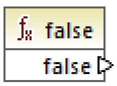


6.6.13 xpath2 | boolean functions

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.

6.6.13.1 false

Returns the Boolean value **false**.

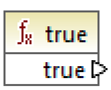


Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.13.2 true

Returns the Boolean value **true**.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.14 xpath2 | constructors

The functions in the "constructors" sub-library of the XPath 2.0 library construct specific data types from the input text. The following table lists the available constructor functions.

xs:ENTITY	**xs:double**	**xs:nonPositiveInteger**
xs:ID	**xs:duration**	**xs:normalizedString**
xs:IDREF	**xs:float**	**xs:positiveInteger**
xs:NCName	**xs:gDay**	**xs:short**
xs:NMTOKEN	**xs:gMonth**	**xs:string**
xs:Name	**xs:gMonthDay**	**xs:time**

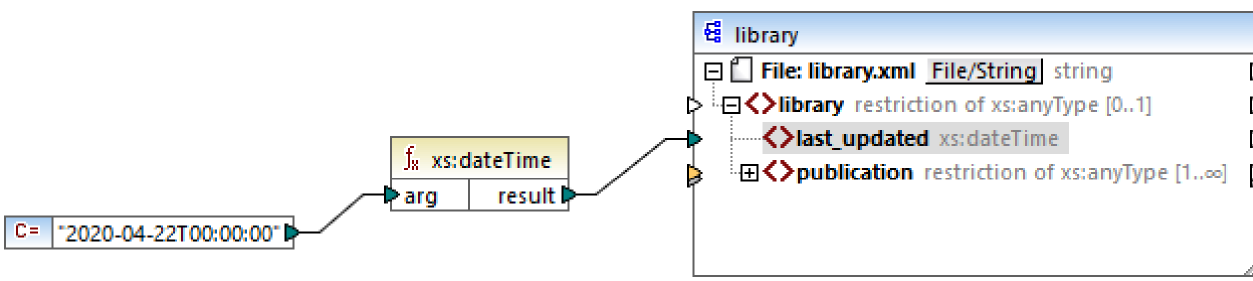
xs:QName	**xs:gYear**	**xs:token**
xs:anyURI	**xs:gYearMonth**	**xs:unsignedByte**
xs:base64Binary	**xs:hexBinary**	**xs:unsignedInt**
xs:boolean	**xs:int**	**xs:unsignedLong**
xs:byte	**xs:integer**	**xs:unsignedShort**
xs:date	**xs:language**	**xs:untypedAtomic**
xs:dateTime	**xs:long**	**xs:yearMonthDuration**
xs:dayTimeDuration	**xs:negativeInteger**	
xs:decimal	**xs:nonNegativeInteger**	

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Example

Typically, the lexical format of the input text must be the one expected of the data type to be constructed. Otherwise, the transformation will not be successful. For example, to construct an `xs:dateTime` value using the **`xs:dateTime`** constructor function, the input text must have the lexical format of the `xs:dateTime` data type, which is **`YYYY-MM-DDTHH:mm:ss`**.



In the mapping illustrated above, a string constant (**`"2020-04-28T00:00:00"`**) has been used to provide the input argument of the function. The input could also have been obtained from an item in the source document. The **`xs:dateTime`** function returns the value **`2020-04-28T00:00:00`** of type `xs:dateTime`.

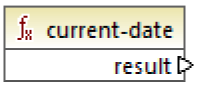
To view the expected data type of a mapping item (including the data type of function arguments), move the mouse cursor over the respective input or output connector.

6.6.15 xpath2 | context functions

The context functions from the **xpath2** library provide miscellaneous information about the current date and time, the default collation used by the processor, the size of the current sequence, and the position of the current node.

6.6.15.1 current-date

Returns the current date (`xs:date`) from the system clock.

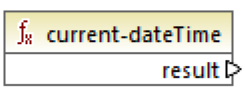


Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.15.2 current-dateTime

Returns the current date and time (`xs:dateTime`) from the system clock.

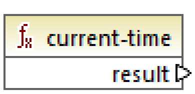


Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.15.3 current-time

Returns the current time (`xs:time`) from the system clock.



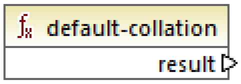
Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.15.4 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.

Comparisons, including for the **`max-string`** and **`min-string`** functions, are based on the default collation.

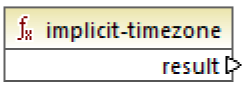


Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.15.5 implicit-timezone

Returns the value of the "implicit timezone" property from the evaluation context.

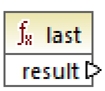


Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.15.6 last

Returns the number of items in the sequence of items currently being processed. Importantly, the sequence of items is determined by the current mapping context[399], as described in the example below.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

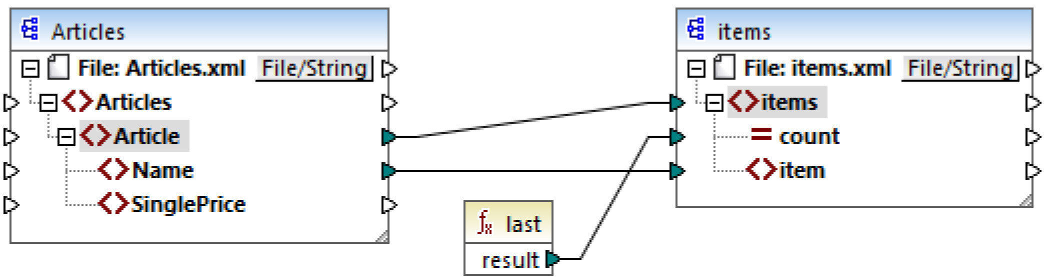
Example

Let's suppose that you have the following source XML file:

```
<Articles>
   <Article>
      <Name>T-Shirt</Name>
      <SinglePrice>25</SinglePrice>
   </Article>
   <Article>
      <Name>Socks</Name>
      <SinglePrice>2.30</SinglePrice>
```

```
   </Article>
   <Article>
      <Name>Jacket</Name>
      <SinglePrice>57.50</SinglePrice>
   </Article>
</Articles>
```

Your goal is to copy data to an XML file with a different schema. Also, the count of all items must be saved to the target XML file. This can be achieved by a mapping like the one below:



In the example above, the **last** function returns the position of the last node in the current parent context and populates the **count** attribute with value **3**.

```
<items count="3">
   <item>T-Shirt</item>
   <item>Pants</item>
   <item>Jacket</item>
</items>
```

Note that value **3** is the position of the last item (and thus the count of all items) in the mapping context created by the connection between **Article** and **items**. If this connection did not exist, items would still be copied to the target, but the **last** function would return value **1** incorrectly, because it would have no parent context [404] to iterate over. (More precisely, it would use the default implicit context created between the root items of both components, which produces a sequence of 1 item, not 3 as expected).

It is generally advisable to use the **count** [229] function from the **core** library instead of the **last** function, because the former has a **parent-context** argument, which enables you to alter the mapping context explicitly.

6.6.16 xpath2 | durations, date and time functions

The duration, date and time functions from the **xpath2** library enable you to adjust the time zone in date and time values, extract particular components from date, time, and duration values, and subtract date and time values.

Adjusting the time zone

To adjust the time zone in date and time values, the following functions are available:

- **adjust-date-to-timezone**

- **`adjust-date-to-timezone`** (with timezone argument)
- **`adjust-dateTime-to-timezone`**
- **`adjust-dateTime-to-timezone`** (with timezone argument)
- **`adjust-time-to-timezone`**
- **`adjust-time-to-timezone`** (with timezone argument)

Each of these related functions takes an `xs:date`, `xs:time`, or `xs:dateTime` value as the first argument and adjusts the input by adding, removing, or modifying the time zone component depending on the value of the second argument (if one is present).

The following situations are possible when the first argument contains no time zone (for example, the date `2020-01` or the time `14:00:00`).

- If the **timezone** argument is present, the result will contain the time zone specified in the second argument. The time zone in the second argument is added.
- If the **timezone** argument is absent, the result will contain the implicit timezone, which is the system's time zone. The system's time zone is added.
- If the **timezone** argument is empty, the result will contain no time zone.

The following situations are possible when the first argument contains a time zone (for example, the date `2020-01-01+01:00` or the time `14:00:00+01:00`).

- If the **timezone** argument is present, the result will contain the time zone specified in the second argument. The original time zone is replaced by the timezone in the second argument.
- If the **timezone** argument is absent, the result will contain the implicit time zone, which is the system's time zone. The original time zone is replaced by the system's time zone.
- If the **timezone** argument is empty, the result will contain no time zone.

Extracting components of dates and times

To extract numeric values such as hours, minutes, days, months, and so on from date and time values, the following functions are available:

- **`day-from-date`**
- **`day-from-dateTime`**
- **`hours-from-dateTime`**
- **`hours-from-time`**
- **`minutes-from-dateTime`**
- **`minutes-from-time`**
- **`month-from-date`**
- **`month-from-dateTime`**
- **`seconds-from-dateTime`**
- **`seconds-from-time`**
- **`timezone-from-date`**
- **`timezone-from-dateTime`**
- **`timezone-from-time`**
- **`year-from-date`**
- **`year-from-dateTime`**

Each of these functions extracts a particular component from `xs:date`, `xs:time`, `xs:dateTime`, and `xs:duration` values. The result will be either `xs:integer` or `xs:decimal`.

Extracting components of durations

To extract time components from durations, the following functions are available:

- **days-from-duration**
- **hours-from-duration**
- **minutes-from-duration**
- **months-from-duration**
- **seconds-from-duration**
- **years-from-duration**

The duration must be specified either as `xs:yearMonthDuration` (for extracting years and months) or `xs:dayTimeDuration` (for extracting days, hours, minutes, and seconds). All functions returns a result of type `xs:integer`, with the exception of the **seconds-from-duration** function, which returns `xs:decimal`.

Subtracting date and time values

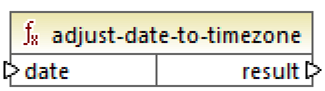
To subtract date and time values, the following functions are available:

- **subtract-dateTimes**
- **subtract-dates**
- **subtract-times**

Each of the subtraction functions enables you to subtract one time value from another and return a duration value.

6.6.16.1 adjust-date-to-timezone

Adjusts an `xs:date` value to the implicit time zone in the evaluation context (the system's time zone).



Languages

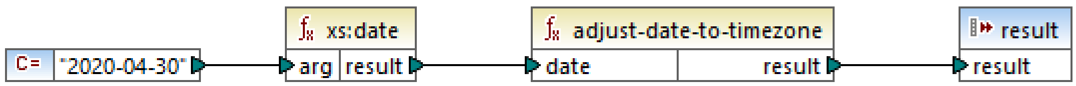
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
date	**xs:date**	The input value of type `xs:date`.

Example

The following mapping constructs an `xs:date` from a string and supplies it as argument to the **adjust-date-to-timezone** function.

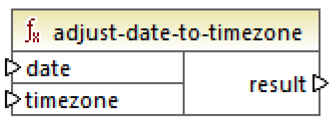


XSLT 2.0 mapping

If the mapping runs on a computer where the system time zone is +02:00, the function adjusts the date value to include the system's time zone. Consequently, the mapping output is **`2020-04-30+02:00`**.

6.6.16.2 adjust-date-to-timezone

Adjusts an `xs:date` value to a specific time zone, or to no time zone at all. If the **timezone** argument is an empty sequence, the function returns an `xs:date` without a time zone. Otherwise, it returns an `xs:date` with a time zone.



Languages

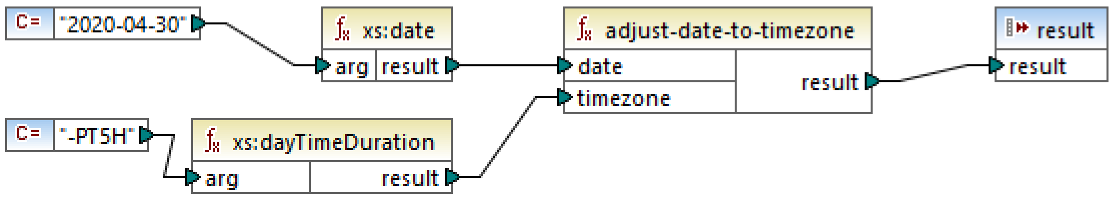
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
date	**`xs:date`**	The input value of type `xs:date`.
timezone	**`xs:dayTimeDuration`**	The time zone expressed as an `xs:dayTimeDuration` value. The value can be negative. For example, a time zone value of -5 hours can be expressed as **`-PT5H`**.

Example

The following mapping constructs both parameters to the **`adjust-date-to-timezone`** function from strings, using the corresponding XPath 2 constructor [310] functions. The goal of the mapping is to adjust the time zone to -5 hours. This time zone can be expressed as **`-PT5H`**.

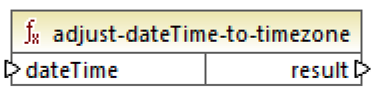


XSLT 2.0 mapping

The function adjusts the date value to the time zone supplied as argument. Consequently, the mapping output is **`2020-04-30-05:00`**.

6.6.16.3 adjust-dateTime-to-timezone

Adjusts an `xs:dateTime` value to the implicit time zone in the evaluation context (the system's time zone).



Languages

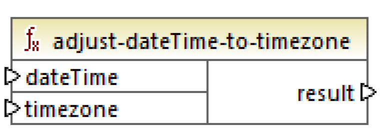
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
dateTime	**`xs:dateTime`**	The input value of type `xs:dateTime`.

6.6.16.4 adjust-dateTime-to-timezone

Adjusts an `xs:dateTime` value to a specific time zone, or to no time zone at all. If the **timezone** argument is an empty sequence, the function returns an `xs:dateTime` without a time zone. Otherwise, it returns an `xs:dateTime` with a time zone.



Languages

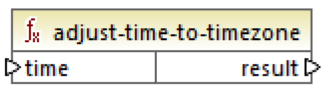
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
dateTime	`xs:dateTime`	The input value of type `xs:dateTime`.
timezone	`xs:dayTimeDuration`	The time zone expressed as an `xs:dayTimeDuration` value. The value can be negative. For example, a time zone value of -5 hours can be expressed as **`-PT5H`**.

6.6.16.5 adjust-time-to-timezone

Adjusts an `xs:time` value to the implicit time zone in the evaluation context (the system's time zone).



Languages

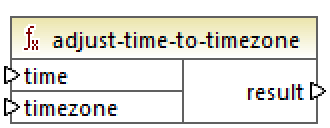
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
time	`xs:time`	The input value of type `xs:time`.

6.6.16.6 adjust-time-to-timezone

Adjusts an `xs:time` value to a specific time zone, or to no time zone at all. If the **timezone** argument is an empty sequence, the function returns an `xs:time` without a time zone. Otherwise, it returns an `xs:time` with a time zone.



Languages

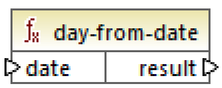
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
time	**xs:time**	The input value of type `xs:time`.
timezone	**xs:dayTimeDuration**	The time zone expressed as an `xs:dayTimeDuration` value. The value can be negative. For example, a time zone value of -5 hours can be expressed as **`-PT5H`**.

6.6.16.7 day-from-date

Returns an `xs:integer` representing the day part of the `xs:date` value supplied as argument.



Languages

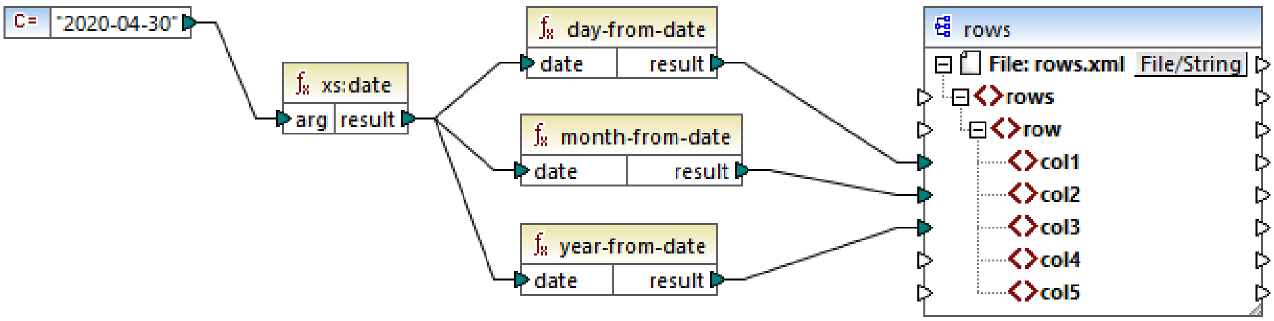
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
date	**xs:date**	The input value of type `xs:date`.

Example

The following mapping converts a string to `xs:date` using the **`xs:date`** constructor function. The **`day-from-date`**, **`month-from-date`**, and **`year-from-date`** functions each extract the respective part of the date and write it to a separate item in the target XML file.



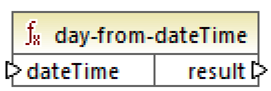
XQuery 1.0 mapping

The mapping output is as follows:

```
<rows>
   <row>
      <col1>30</col1>
      <col2>4</col2>
      <col3>2020</col3>
   </row>
</rows>
```

6.6.16.8 day-from-dateTime

Returns an `xs:integer` representing the day part of the `xs:dateTime` value supplied as argument.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
dateTime	`xs:dateTime`	The input value of type `xs:dateTime`.

6.6.16.9 days-from-duration

Returns an `xs:integer` representing the "days" component of the canonical representation of the duration value supplied as argument.

Languages

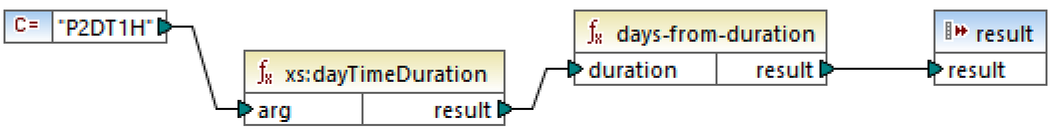
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
duration	`xs:duration`	The input value of type `xs:duration`.

Example

The mapping illustrated below constructs the `xs:dayTimeDuration` of **`P2DT1H`** (2 days and 1 hours) and supplies it as input to the **`days-from-duration`** function. The result is **2**.



XSLT 2.0 mapping

Note: If the duration is **`P1DT24H`** (1 day and 24 hours), the function returns **2**, not **1**. This is because the canonical representation of **`P1DT24H`** is actually **`P2D`** (2 days).

6.6.16.10 hours-from-dateTime

Returns an `xs:integer` representing the hours part of the `xs:dateTime` value supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
dateTime	`xs:dateTime`	The input value of type `xs:dateTime`.

6.6.16.11 hours-from-duration

Returns an `xs:integer` representing the hours component of the canonical representation of the duration value supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
duration	`xs:duration`	The input value of type `xs:duration`.

Example

If the duration is **`PT1H60M`** (1 hour and 60 minutes), the function returns **2**, not **1**. This is because the canonical representation of **`PT1H60M`** is actually **`PT2H`** (2 hours).

6.6.16.12 hours-from-time

Returns an `xs:integer` representing the hours part of the `xs:time` value supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
time	`xs:time`	The input value of type `xs:time`.

6.6.16.13 minutes-from-dateTime

Returns an `xs:integer` representing the minutes part of the `xs:dateTime` supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
dateTime	`xs:dateTime`	The input value of type `xs:dateTime`.

6.6.16.14 minutes-from-duration

Returns an `xs:integer` representing the minutes component of the canonical representation of the duration supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
duration	**xs:duration**	The input value of type `xs:duration`.

Example

If the duration is **`PT1M60S`** (1 minute and 60 seconds), the function returns **2**, not **1**. This is because the canonical representation of **`PT1M60S`** is actually **`PT2M`** (2 minutes).

6.6.16.15 minutes-from-time

Returns an `xs:integer` representing the minutes part of the `xs:time` value supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
time	**xs:time**	The input value of type `xs:time`.

6.6.16.16 month-from-date

Returns an `xs:integer` representing the month part of the `xs:date` value supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
date	**xs:date**	The input value of type `xs:date`.

6.6.16.17 month-from-dateTime

Returns an `xs:integer` representing the month part of the `xs:dateTime` value supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
dateTime	**xs:dateTime**	The input value of type `xs:dateTime`.

6.6.16.18 months-from-duration

Returns an `xs:integer` representing the months component in the canonical representation of the duration value supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
duration	**xs:duration**	The input value of type `xs:duration`.

6.6.16.19 seconds-from-dateTime

Returns an xs:integer representing the seconds component in the localized value of dateTime.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
dateTime	**xs:dateTime**	

6.6.16.20 seconds-from-duration

Returns an `xs:integer` representing the seconds component in the canonical representation of the duration value supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
duration	**xs:duration**	The input value of type `xs:duration`.

6.6.16.21 seconds-from-time

Returns an `xs:integer` representing the seconds part of the `xs:time` value supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
time	**xs:time**	The input value of type `xs:time`.

6.6.16.22 subtract-dateTimes

Returns the `xs:dayTimeDuration` that corresponds to the difference between the normalized value of **dateTime1** and the normalized value of **dateTime2**.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
dateTime1	`xs:dateTime`	The first input value.
dateTime2	`xs:dateTime`	The second input value.

6.6.16.23 subtract-dates

Returns the `xs:dayTimeDuration` that corresponds to the difference between the normalized value of **date1** and the normalized value of **date2**.

Languages

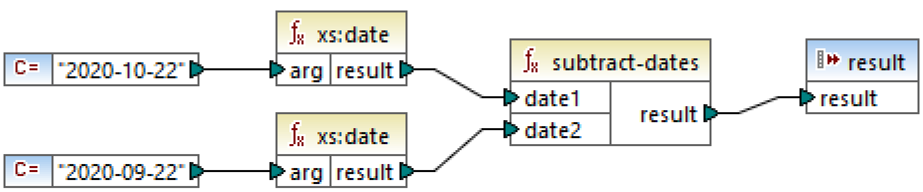
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
date1	`xs:date`	The first input value.
date2	`xs:date`	The second input value.

Example

The mapping illustrated below subtracts two dates (2020-10-22 minus 2020-09-22). The result is the value **`P30D`** of type `xs:dayTimeDuration`., which represents a duration of 30 days.



6.6.16.24 subtract-times

Returns the `xs:dayTimeDuration` that corresponds to the difference between the normalized value of **time1** and the normalized value of **time2**.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
time1	**xs:time**	The first input value.
time2	**xs:time**	The second input value.

6.6.16.25 timezone-from-date

Returns the timezone component of the date supplied as argument. The result is an `xs:dayTimeDuration` that indicates deviation from UTC; its value may range from +14:00 to -14:00 hours, both inclusive.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
date	**xs:date**	The input value of type `xs:date`.

6.6.16.26 timezone-from-dateTime

Returns the timezone component of the `xs:dateTime` value supplied as argument. The result is an `xs:dayTimeDuration` that indicates deviation from UTC; its value may range from +14:00 to -14:00 hours, both inclusive.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
dateTime	**xs:dateTime**	The input value of type `xs:dateTime`.

6.6.16.27 timezone-from-time

Returns the timezone component of the `xs:time` value supplied as argument. The result is an `xs:dayTimeDuration` that indicates deviation from UTC; its value may range from +14:00 to -14:00 hours, both inclusive.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
time	**xs:time**	The input value of type xs:time.

6.6.16.28 year-from-date

Returns an xs:integer representing the year part of the xs:date value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
date	**xs:date**	The input value of type xs:date.

6.6.16.29 year-from-dateTime

Returns an xs:integer representing the year part of the xs:dateTime value supplied as argument.

Languages
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
dateTime	**xs:dateTime**	The input value of type xs:dateTime.

6.6.16.30 years-from-duration

Returns an xs:integer representing the years component in the canonical lexical representation of the duration value supplied as argument.

Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
duration	`xs:duration`	The input value of type `xs:duration`.

6.6.17 xpath2 | node functions

The node functions from the **xpath2** library provide information about nodes (items) on a mapping component.

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.

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 (see the example given for the **local-name**[332] function). Each of these three functions has two variants:

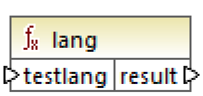
- With no argument: the function is then applied to the context node (for an example of a context node, see the example given for the **lang**[330] function).
- With an argument that must be a node: the function is applied to the connected node.

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. 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**[330] function).
- With an argument that must be a node: the function is applied to the connected node.

6.6.17.1 lang

Returns **true** if the context node has an `xml:lang` attribute with a value that either matches exactly the **testlang** argument, or is a subset of it. Otherwise, the function returns **false**.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

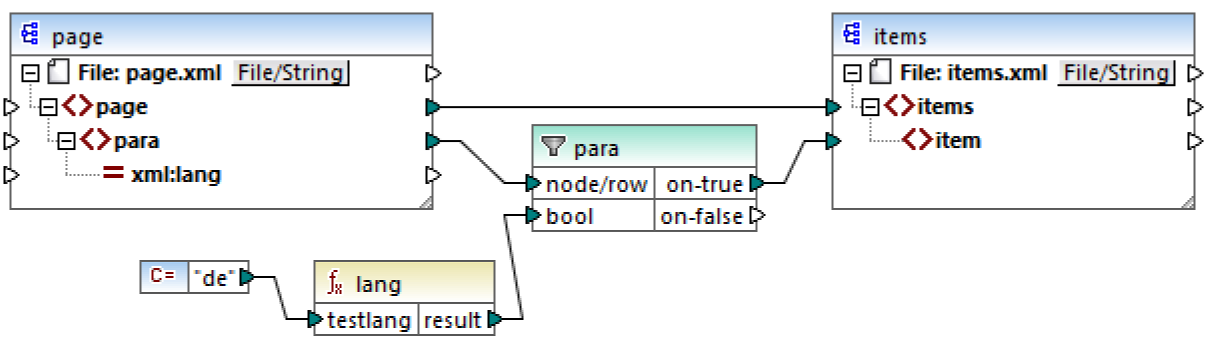
Name	Type	Description
testlang	xs:string	The language code to check, for example, "en".

Example

The following XML contains **para** elements with different values for the `xml:lang` attribute.

```
<page>
   <para xml:lang="en">Good day!</para>
   <para xml:lang="fr">Bonjour!</para>
   <para xml:lang="de-AT">Grüss Gott!</para>
   <para xml:lang="de-DE">Guten Tag!</para>
   <para xml:lang="de-CH">Grüezi!</para>
</page>
```

The mapping illustrated below filters only the German paragraphs, regardless of the country variant, with the help of the **lang** function.



XSLT 2.0 mapping

In the mapping above, for each **para** in the source, an **item** is created in the target, conditionally. The condition is provided by a filter which passes on to the target only those nodes where the **lang** function returns **true**. That is, only those nodes that have the `xml:lang` attribute set to "de" (or a subset of "de") will satisfy the filter's condition. Consequently, the mapping output is as follows:

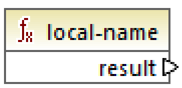
```
<items>
   <item>Grüss Gott!</item>
   <item>Guten Tag!</item>
</items>
```

```
  <item>Grüezi!</item>
</items>
```

Note that the **lang** function operates in the context of each **para**, because of the parent connection between **para** and **item**, see also The Mapping Context [399].

6.6.17.2 local-name

Returns the local part of the name of the context node as an `xs:string`. This is a parameterless variant of the **local-name** function where the context node is determined by the connections in your mapping. To specify a node explicitly, use the **local-name** [332] function that takes an input node as parameter.

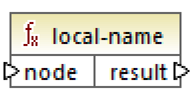


Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.17.3 local-name

Returns the local part of the name of **node** as an `xs:string`.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
node	**node()**	The input node.

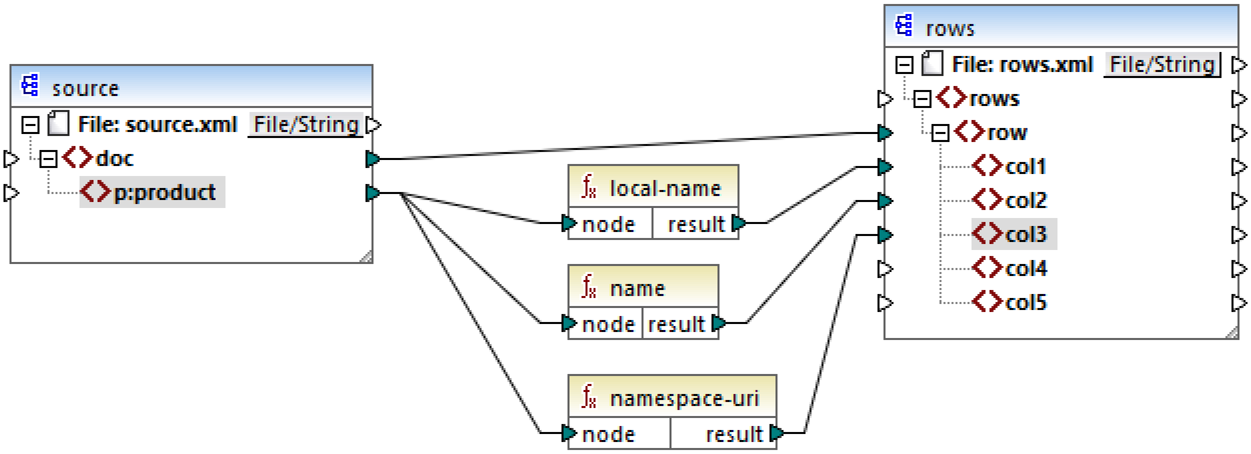
Example

In the following XML file, the name of the `p:product` element is a prefixed qualified name (QName). The prefix "p" is mapped to the namespace "http://mycompany.com".

```
<?xml version="1.0" encoding="UTF-8"?>
<doc xmlns:p="http://mycompany.com" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:noNamespaceSchemaLocation="source.xsd">
   <p:product/>
</doc>
```

The following mapping extracts the local name, the name, and the namespace URI of the node and writes these values to a target file:



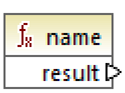
XSLT 2.0 mapping

The mapping output is displayed below. Each **col** item lists the result of the **local-name**, **name**, and **namespace-uri** functions, respectively.

```
<rows>
   <row>
      <col1>product</col1>
      <col2>p:product</col2>
      <col3>http://mycompany.com</col3>
   </row>
</rows>
```

6.6.17.4 name

Returns the name of the context node. This is a parameterless variant of the **name** function where the context node is determined by the connections in your mapping. To specify a node explicitly, use the **name** 334 function that takes an input node as parameter.

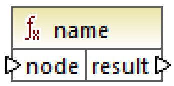


Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.17.5 name

Returns the name of a node.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

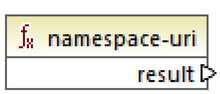
Name	Type	Description
node	`node()`	The input node.

Example

See the example given for the `local-name` [332] function.

6.6.17.6 namespace-uri

Returns the namespace URI of the QName of the context node, as an `xs:string`. This is a parameterless variant of the **`namespace-uri`** function where the context node is determined by the connections in your mapping. To specify a node explicitly, use the `namespace-uri` [334] function that takes an input node as parameter.

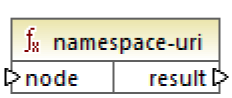


Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.17.7 namespace-uri

Returns the namespace URI of the QName of **node**, as an `xs:string`.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
node	`node()`	The input node.

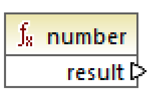
Example

See the example given for the **`local-name`** [332] function.

6.6.17.8 number

Returns the value of the context node, converted to an `xs:double`. This is a parameterless variant of the **`number`** function where the context node is determined by the connections in your mapping. To specify a node explicitly, use the **`number`** [335] function that takes an input node as parameter.

The only types that can be converted to numbers are Booleans, numeric strings, and other numeric types. Non-numeric input values (such as a non-numeric string) result in NaN (Not a Number).

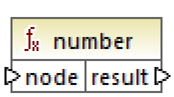


Languages

XQuery, XSLT 2.0, XSLT 3.0.

6.6.17.9 number

Returns the value of **node**, converted to an `xs:double`. The only types that can be converted to numbers are Booleans, numeric strings, and other numeric types. Non-numeric input values (such as a non-numeric string) result in NaN (Not a Number).



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

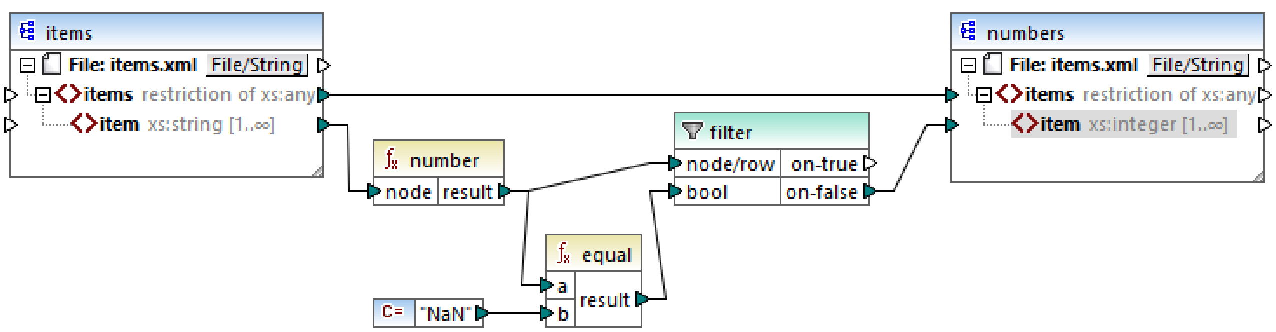
Name	Type	Description
node	`mf:atomic`	The input node.

Example

The following XML contains items of type `string`:

```
<items>
   <item>1</item>
   <item>2</item>
   <item>Jingle Bells</item>
</items>
```

The mapping illustrated below attempts to convert all these strings to numeric values and write them to a target XML file. Notice that the data type of **item** in the target XML component is `xs:integer` while the source **item** is of `xs:string` data type. If the conversion is not successful, the item must be skipped and not copied to the target file.



XSLT 2.0 mapping

To achieve the mapping goal, a filter was used. The **`equal`** function checks if the result of the conversion is "NaN". If this is false, this indicates a successful conversion, so the item is copied to the target. The output of the mapping is as follows:

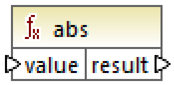
```
<items>
   <item>1</item>
   <item>2</item>
</items>
```

6.6.18 xpath2 | numeric functions

The numeric functions of the **xpath2** library include the **`abs`** and **`round-half-to-even`** functions.

6.6.18.1 abs

Returns the absolute value of the argument. For example, if the input argument is **-2** or **2**, the function returns **2**.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

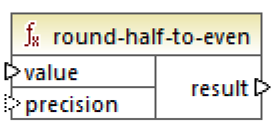
Parameters

Name	Type	Description
value	`xs:decimal`	The input value.

6.6.18.2 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 **2.142**. 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**.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:decimal`	Mandatory argument which provides the input value to be rounded.

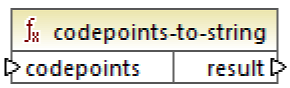
Name	Type	Description
precision	`xs:integer`	Optional argument which specifies the number of decimal places to round to. The default value is **0**.

6.6.19 xpath2 | string functions

The string functions of the **xpath2** library enable you to process strings (this includes comparing strings, converting strings to upper or lower case, extracting substrings from strings, and others).

6.6.19.1 codepoints-to-string

Creates a string from a sequence of Unicode code points. This function is the opposite of the **string-to-codepoints** 346 function.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
codepoints	`ZeroOrMore xs:integer`	This input must be connected to a sequence of items of integer type, where each integer specifies a Unicode code point.

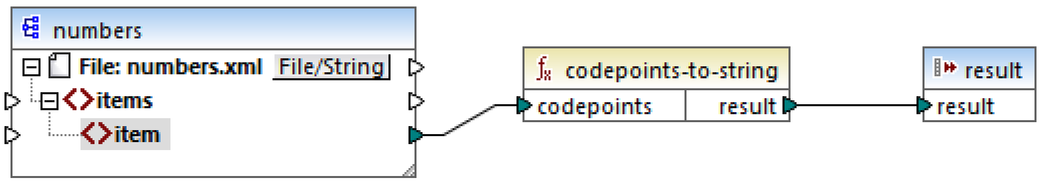
Example

The following XML contains multiple **item** elements that store each Unicode code point values.

```
<items>
   <item>77</item>
   <item>97</item>
   <item>112</item>
   <item>70</item>
   <item>111</item>
   <item>114</item>
   <item>99</item>
```

```
   <item>101</item>
</items>
```

The mapping illustrated below supplies the sequence of items as argument to the **codepoint-to-string** function.



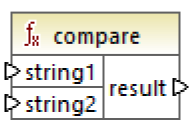
XSLT 2.0 mapping

The mapping output is **MapForce**.

6.6.19.2 compare

The **compare** function takes two strings as arguments and compares them for equality and alphabetically. If **string1** is alphabetically less than **string2** (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 **string1** is greater than **string2** (for example, "B" and "A"), then the function returns **1**.

This variant of the function uses the default collation, which is Unicode. Another variant 340 of this function exists where you can supply the collation as argument.



Languages

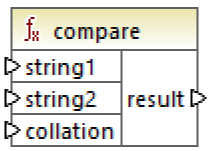
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
string1	xs:string	The first input string.
string2	xs:string	The second input string.

6.6.19.3 compare

The `compare` function takes two strings as arguments and compares them for equality and alphabetically, using the collation supplied as argument. If **string1** is alphabetically less than **string2** (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 **string1** is greater than **string2** (for example, "B" and "A"), then the function returns **1**.



Languages

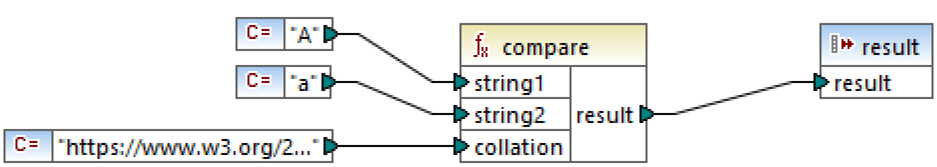
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
string1	`xs:string`	The first input string.
string2	`xs:string`	The second input string.
collation	`xs:string`	Specifies the collation to use for string comparison. This input may originate from the output of the `default-collation`[312] function or it may be a collation such as `http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive`.

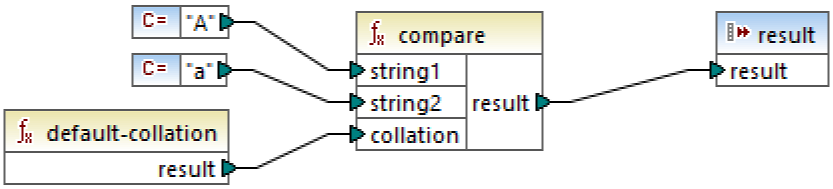
Example

The following mapping compares the strings "A" and "a" using the case insensitive collation `http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive`, which is supplied by a constant.



XSLT 2.0 Mapping

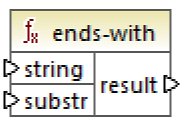
The result of the mapping above is **0**, meaning that both strings are treated as equal. However, if you replace the collation with the one provided by the **default-collation** function, the collation changes to the default Unicode code point collation, and the mapping result becomes **-1** ("A" is alphabetically less than "a").



6.6.19.4 ends-with

Returns **true** if **string** ends with **substr**; **false** otherwise. The returned value is of type `xs:boolean`.

This variant of the function uses the default collation, which is Unicode. Another variant[342] of this function exists where you can supply the collation as argument.



Languages

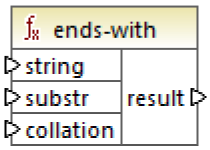
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
string	`xs:string`	The input string (that is, the "haystack").
substr	`xs:string`	The substring (that is, the "needle").

6.6.19.5 ends-with

Returns **true** if **string** ends with **substr**; **false** otherwise. The returned value is of type `xs:boolean`.



Languages

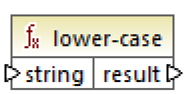
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
string	`xs:string`	The input string (that is, the "haystack").
substr	`xs:string`	The substring (that is, the "needle").
collation	`xs:string`	Specifies the collation to use for string comparison. This input may originate from the output of the **`default-collation`** 312 function or it may be a collation such as `http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive`.

6.6.19.6 lower-case

Returns the value of **string** after translating every character to its lower-case correspondent.



Languages

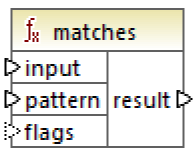
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
string	`xs:string`	The input value.

6.6.19.7 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.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

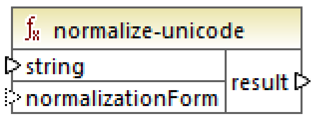
Parameters

Name	Type	Description
input	`xs:string`	The input string.
pattern	`xs:string`	The regular expression to match, see Regular Expressions [221].
flags	`xs:string`	Optional argument that influences the matching. This argument may supply any combination of the following flags: `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 four flags are as follows: `i` Use case-insensitive mode. The default is case-sensitive. `m` Use multi-line 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.

Name	Type	Description
		x Ignore whitespace. By default, whitespace characters are not ignored.

6.6.19.8 normalize-unicode

Returns the value of **string** normalized according to the rules of the normalization form specified (the second argument). For more information about Unicode normalization, see §2.2 of https://www.w3.org/TR/charmod-norm/.



Languages

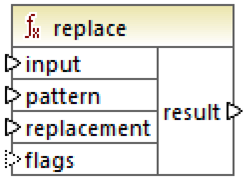
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
string	xs:string	The string value to be normalized.
normalizationForm	xs:string	Optional argument which supplies the normalization form. The default is Unicode Normalization Form C (NFC). The normalization forms NFC, NFD, NFKC, and NFKD are supported.

6.6.19.9 replace

This function takes an input string, a regular expression, and a replacement string as arguments. It replaces all matches of the regular expression in the input string with the replacement string. If the regular expression matches two overlapping strings in the input string, only the first match is replaced.



Languages

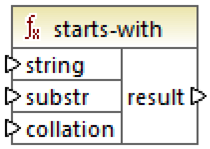
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
input	`xs:string`	The input string.
pattern	`xs:string`	The regular expression to match, see Regular Expressions[221].
replacement	`xs:string`	The replacement string.
flags	`xs:string`	Optional argument that influences the matching. This argument is used in the same way as the **flags** argument of the **`matches`**[343] function.

6.6.19.10 starts-with

Returns **true** if **string** starts with **substr**; **false** otherwise. The returned value is of type `xs:boolean`. String comparison takes place according to the specified collation.



Languages

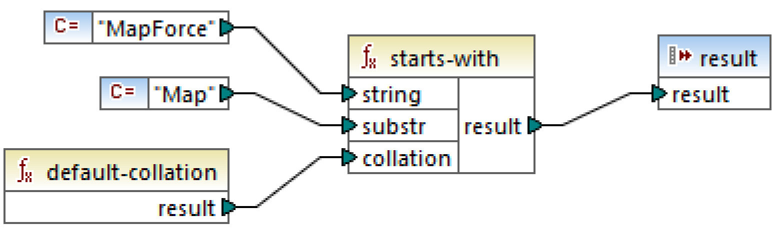
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
string	`xs:string`	The input string (that is, the "haystack").
substr	`xs:string`	The substring (that is, the "needle").
collation	`xs:string`	Specifies the collation to use for string comparison. This input may originate from the output of the **default-collation** [312] function or it may be a collation such as `http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive`.

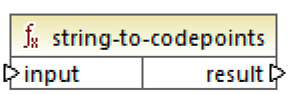
Example

The following mapping returns the value `true`, because the input string "MapForce" begins with the substring "Map", assuming that the default Unicode collation is used.



6.6.19.11 string-to-codepoints

Returns the sequence of Unicode code points (integer values) that constitute the string supplied as argument. This function is the opposite of the **codepoints-to-string** [338] function.



Languages

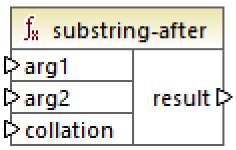
XQuery, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
input	`xs:string`	The input string

6.6.19.12 substring-after

Returns the part of string **arg1** that occurs after the string **arg2**.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

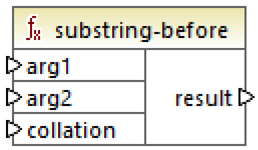
Name	Type	Description
arg1	`xs:string`	The input string (that is, the "haystack").
arg2	`xs:string`	The substring (that is, the "needle").
collation	`xs:string`	Specifies the collation to use for string comparison. This input may originate from the output of the `default-collation` [312] function or it may be a collation such as `http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive`.

Example

If **arg1** is "MapForce", **arg2** is "Map", and **collation** is `default-collation` [312], the function returns "Force".

6.6.19.13 substring-before

Returns the part of string **arg1** that occurs before the string **arg2**.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

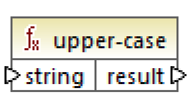
Name	Type	Description
arg1	`xs:string`	The input string (that is, the "haystack").
arg2	`xs:string`	The substring (that is, the "needle").
collation	`xs:string`	Specifies the collation to use for string comparison. This input may originate from the output of the `default-collation`[312] function or it may be a collation such as `http://www.w3.org/2005/xpath-functions/collation/html-ascii-case-insensitive`.

Example

If **arg1** is "MapForce", **arg2** is "Force", and **collation** is `default-collation`[312], the function returns "Map".

6.6.19.14 upper-case

Returns the value of **string** after translating every character to its upper-case correspondent.



Languages

XQuery, XSLT 2.0, XSLT 3.0.

Parameters

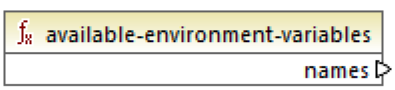
Name	Type	Description
string	`xs:string`	The input string.

6.6.20 xpath3 | external information functions

The external information functions of the **xpath3** library enable you to obtain information about the XSLT execution environment or retrieve data from external resources.

6.6.20.1 available-environment-variables

Returns a list of environment variable names that are suitable for passing to the **`environment-variable`** function, as a (possibly empty) sequence of strings.

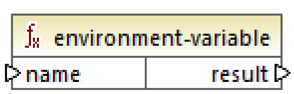


Languages

XSLT 3.0.

6.6.20.2 environment-variable

Returns the value of a system environment variable, if it exists. The return type is `xs:string`.



Languages

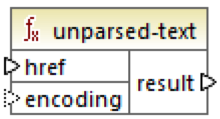
XSLT 3.0.

Parameters

Name	Type	Description
name	`xs:string`	The name of the environment variable.

6.6.20.3 unparsed-text

Reads an external resource (for example, a file) and returns a string representation of the resource.



Languages

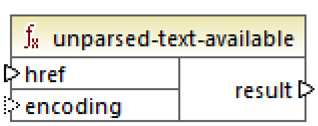
XSLT 3.0.

Parameters

Name	Type	Description
href	`xs:string`	A string in the form of a URI reference.
encoding	`xs:string`	Optional argument. Specifies the name of the encoding, for example "UTF-8", "UTF-16". If the encoding cannot be determined automatically, then UTF-8 is assumed.

6.6.20.4 unparsed-text-available

Determines whether a call to **`unparsed-text`** with particular arguments would succeed. The return type is `xs:boolean`.



Languages

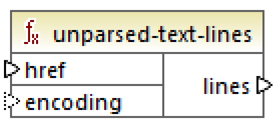
XSLT 3.0.

Parameters

Name	**Type**	**Description**
href	`xs:string`	A string in the form of a URI reference.
encoding	`xs:string`	Optional argument. Specifies the name of the encoding, for example "UTF-8", "UTF-16". If the encoding cannot be determined automatically, then UTF-8 is assumed.

6.6.20.5 unparsed-text-lines

Reads an external resource (for example, a file) and returns its contents as a sequence of strings, one for each line of text in the string representation of the resource.



Languages

XSLT 3.0.

Parameters

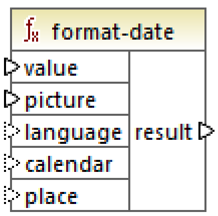
Name	**Type**	**Description**
href	`xs:string`	A string in the form of a URI reference.
encoding	`xs:string`	Optional argument. Specifies the name of the encoding, for example "UTF-8", "UTF-16". If the encoding cannot be determined automatically, then UTF-8 is assumed.

6.6.21 xpath3 | formatting functions

The formatting functions available of the **xpath3** library are used to format date, time and integer values.

6.6.21.1 format-date

Returns a string containing an `xs:date` value formatted for display.



Languages

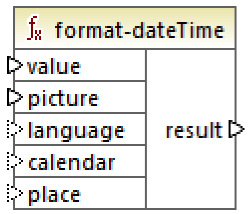
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:date`	The input `xs:date` value to be formatted. Mandatory parameter.
picture	`xs:string`	Mandatory parameter. See section 9.8.4.1 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).
language	`xs:string`	Optional parameter. See section 9.8.4.8 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).
calendar	`xs:string`	Same as above.
place	`xs:string`	Same as above.

6.6.21.2 format-dateTime

Returns a string containing an `xs:dateTime` value formatted for display.



Languages

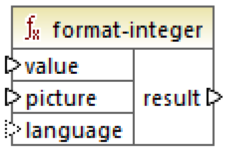
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:dateTime`	The input `xs:dateTime` value to be formatted.
picture	`xs:string`	Mandatory parameter. See section 9.8.4.1 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).
language	`xs:string`	Optional parameter. See section 9.8.4.8 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).
calendar	`xs:string`	Same as above.
place	`xs:string`	Same as above.

6.6.21.3 format-integer

Formats an integer according to a given picture string, using the conventions of a given natural language if specified.



Languages

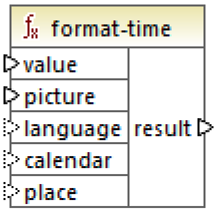
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:integer`	The input integer value to be formatted.
picture	`xs:string`	Mandatory parameter. See section 4.6.1 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).
language	`xs:string`	Optional parameter. Specifies the natural language according to which the value should be formatted. If specified, this value must be either an empty string or any value that would be allowed for the `xml:lang` attribute according to the "Extensible Markup Language (XML) 1.0 W3C Recommendation (https://www.w3.org/TR/xml).

6.6.21.4 format-time

Returns a string containing an `xs:time` value formatted for display.



Languages

XSLT 3.0.

Parameters

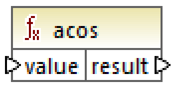
Name	Type	Description
value	`xs:time`	The input `xs:time` value to be formatted.
picture	`xs:string`	Mandatory parameter. See section 9.8.4.1 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).
language	`xs:string`	Optional parameter. See section 9.8.4.8 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).
calendar	`xs:string`	Same as above.
place	`xs:string`	Same as above.

6.6.22 xpath3 | math functions

The math functions of the **xpath3** library are used to perform trigonometric and other mathematical calculations.

6.6.22.1 acos

Returns the arc cosine of an angle, in the range of **0** through **pi**.



Languages

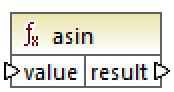
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:double`	The input value.

6.6.22.2 asin

Returns the arc sine of an angle, in the range of **-pi/2** through **pi/2**.



Languages

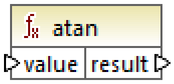
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:double`	The input value.

6.6.22.3 atan

Returns the arc tangent of an angle, in the range of **-pi/2** through **pi/2**.



Languages

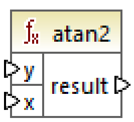
XSLT 3.0.

Parameters

Name	Type	Description
value	**xs:double**	The input value.

6.6.22.4 atan2

Returns the angle in radians subtended at the origin by the point on a plane with coordinates (x, y) and the positive x-axis.



Languages

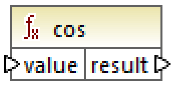
XSLT 3.0.

Parameters

Name	Type	Description
y	**xs:double**	The x coordinate.
x	**xs:double**	The y coordinate.

6.6.22.5 cos

Returns the trigonometric cosine of the angle given by value. The unit of value is radian.



Languages

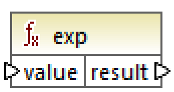
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:double`	The input value.

6.6.22.6 exp

Returns Euler's number *e* raised to the power of the value.



Languages

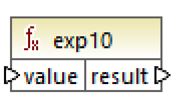
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:double`	The input value.

6.6.22.7 exp10

Returns 10 raised to the power of the value.



Languages

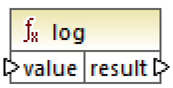
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:double`	The input value.

6.6.22.8 log

Returns the natural logarithm (base e) of a value.



Languages

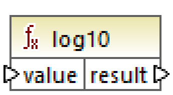
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:double`	The input value.

6.6.22.9 log10

Returns the decimal logarithm (base 10) of a value.



Languages

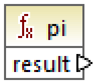
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:double`	The input value.

6.6.22.10 pi

Returns an approximation to the mathematical constant **pi**.

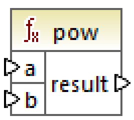


Languages

XSLT 3.0.

6.6.22.11 pow

Returns the value of **a** raised to the power of **b**.



Languages

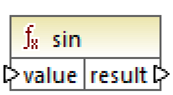
XSLT 3.0.

Parameters

Name	Type	Description
a	`xs:double`	The input value **a**.
b	`xs:double`	The input value **b**.

6.6.22.12 sin

Returns the trigonometric sine of the angle given by value. The unit of value is radian.



Languages

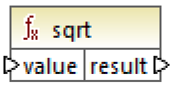
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:double`	The input value.

6.6.22.13 sqrt

Returns the non-negative square root of the argument.



Languages

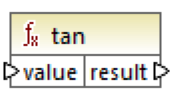
XSLT 3.0.

Parameters

Name	Type	Description
value	`xs:double`	The input value.

6.6.22.14 tan

Returns the trigonometric tangent of the angle given by value. The unit of value is radian.



Languages

XSLT 3.0.

Parameters

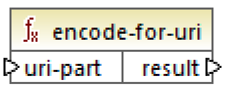
Name	Type	Description
value	`xs:double`	The input value.

6.6.23 xpath3 | URI functions

The URI functions in the **xpath3** library perform encoding, escaping, and conversion of values intended for use in URIs.

6.6.23.1 encode-for-uri

Encodes reserved characters in a string that is intended to be used in the path segment of a URI. For further information about this function, see section 6.2 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).



Languages

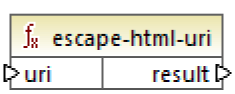
XSLT 3.0.

Parameters

Name	Type	Description
uri-part	xs:string	The input URI value to encode.

6.6.23.2 escape-html-uri

Escapes a URI in the same way that HTML user agents handle attribute values expected to contain URIs. For further information about this function, see section 6.4 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).



Languages

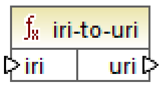
XSLT 3.0.

Parameters

Name	Type	Description
uri	xs:string	The input URI value to escape.

6.6.23.3 iri-to-uri

Converts a string containing an IRI (Internationalized Resource Identifier) into a URI (Uniform Resource Identifier). For further information about this function, see section 6.3 of the "XPath and XQuery Functions and Operators 3.1" W3C Recommendation (https://www.w3.org/TR/xpath-functions-31).



Languages

XSLT 3.0.

Parameters

Name	Type	Description
iri	`xs:string`	The input IRI value.

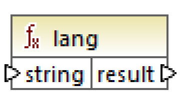
6.6.24 xslt | xpath functions

The functions in this sub-group are XPath 1.0 functions that retrieve information about mapping items (or nodes). Most of these functions take a node as argument and return information about that node. The `last` and `position` functions operate in the current mapping context[399], which is determined by the connections on your mapping.

Note: Additional XPath 1.0 functions can be found in the **core** library.

6.6.24.1 lang

Returns **true** if the context node has an `xml:lang` attribute with a value that either matches exactly the **string** argument, or is a subset of it. Otherwise, the function returns **false**.



Languages

XSLT 1.0.

Parameters

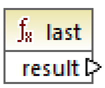
Name	Type	Description
string	`xs:string`	The language code to check, for example, "en".

Example

See the example given for the **lang** [330] function of the **xpath2** library.

6.6.24.2 last

Returns the position number of the last node in the processed node list.



Languages

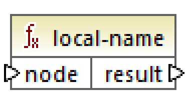
XSLT 1.0.

Example

See the example given for the **last** [313] function of the **xpath2** library.

6.6.24.3 local-name

Returns the local part of the name of the node supplied as argument.



Languages

XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

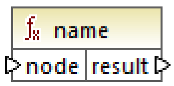
Name	Type	Description
node	`node()`	The input node.

Example

See the example given for the **local-name** 332 function of the **xpath2** library.

6.6.24.4 name

Returns the name of the node supplied as argument.



Languages

XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

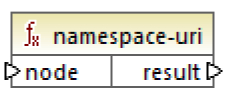
Name	Type	Description
node	node()	The input node.

Example

See the example given for the **local-name** 332 function of the **xpath2** library.

6.6.24.5 namespace-uri

Returns the namespace URI of the node supplied as argument.



Languages

XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

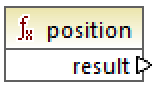
Name	Type	Description
node	node()	The input node.

Example

See the example given for the **`local-name`** [332] function of the **`xpath2`** library.

6.6.24.6 position

Returns the position of the current node in the node set that is being processed.



Languages

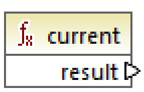
XSLT 1.0.

6.6.25 xslt | xslt functions

The functions in this group are miscellaneous XSLT 1.0 functions.

6.6.25.1 current

The **`current`** function takes no argument and returns the current node.

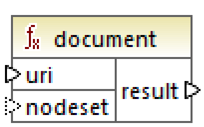


Languages

XSLT 1.0.

6.6.25.2 document

Accesses nodes from an external XML document. The result is output to a node in the output document.



Languages

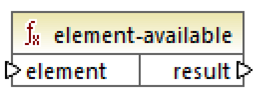
XSLT 1.0.

Parameters

Name	Type	Description
uri	`xs:string`	Mandatory. Specifies the path to the XML document. The XML document must be valid and parseable.
nodeset	`node()`	Optional. Specifies a node, the base URI of which is used to resolve the URI supplied as the first argument if it is relative.

6.6.25.3 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.



Languages

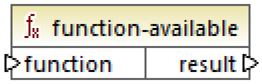
XSLT 1.0.

Parameters

Name	Type	Description
element	`xs:string`	The element name.

6.6.25.4 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.



Languages

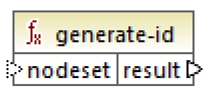
XSLT 1.0.

Parameters

Name	Type	Description
function	`xs:string`	The function name.

6.6.25.5 generate-id

The **`generate-id`** function generates a unique string that identifies the first node in the node set 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.



Languages

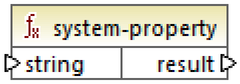
XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
nodeset	`node()`	Optional argument that supplies the input node.

6.6.25.6 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.



Languages

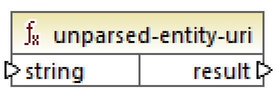
XSLT 1.0, XSLT 2.0, XSLT 3.0.

Parameters

Name	Type	Description
string	`xs:string`	Specifies the property name, which can be any of the following: `xsl:version`, `xsl:vendor`, `xsl:vendor-url`.

6.6.25.7 unparsed-entity-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.



Languages

XSLT 1.0.

Parameters

Name	Type	Description
string	`xs:string`	The name of the unparsed entity whose URI should be retrieved.

7 Advanced Mapping Scenarios

Altova website: Data integration tool

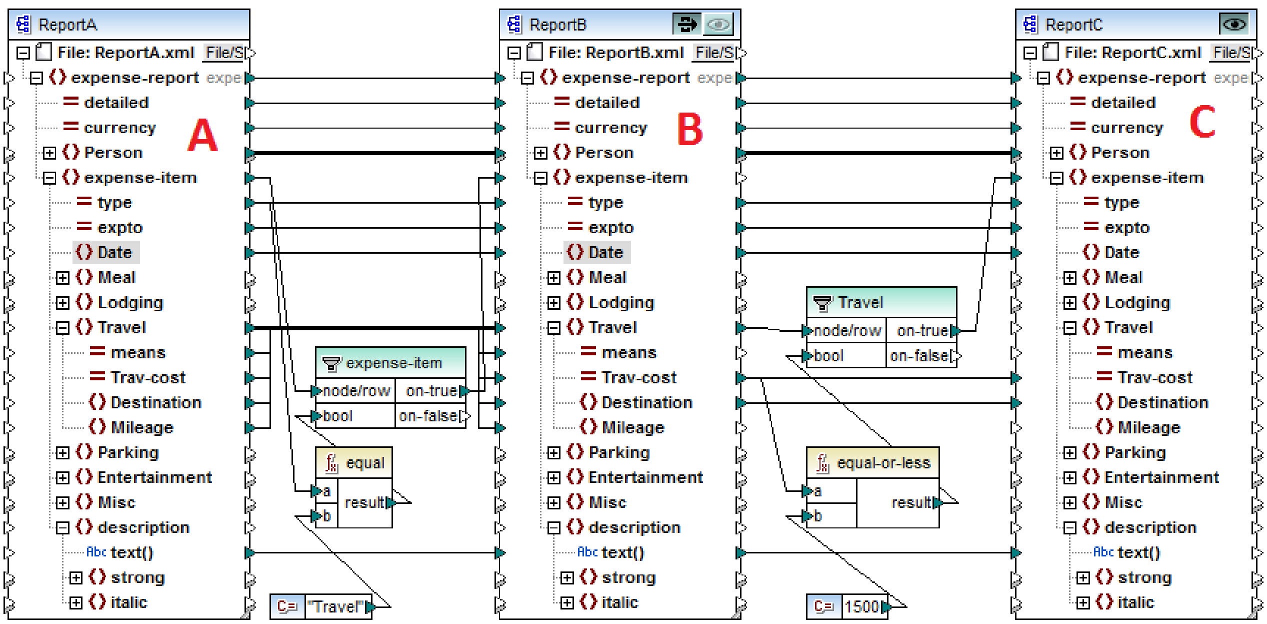
This section describes advanced mapping scenarios and includes the following topics:

- Chained Mappings 371
- Mapping Node Names 380
- Mapping Rules and Strategies 397
- Processing Multiple Input or Output Files 417

7.1 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\MapForce2023\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 417.

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 373 and Example: Pass-Through Inactive 377, illustrate in more detail how the source data is transferred differently when the pass-through button is active or inactive.

7.1.1 Example: Pass-Through Active

The mapping used in this example (**ChainedReports.mfd**) is available in the **<Documents>\Altova\MapForce2023\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:

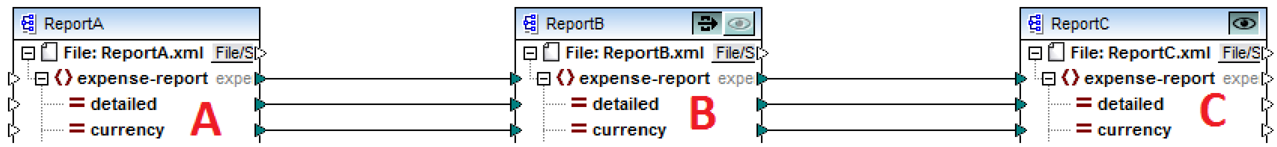
```
<?xml version="1.0" encoding="UTF-8"?>
<expense-report currency="USD" detailed="true">
   <Person>
      <First>Fred</First>
      <Last>Landis</Last>
      <Title>Project Manager</Title>
      <Phone>123-456-78</Phone>
      <Email>f.landis@nanonull.com</Email>
   </Person>
   <expense-item type="Travel" 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

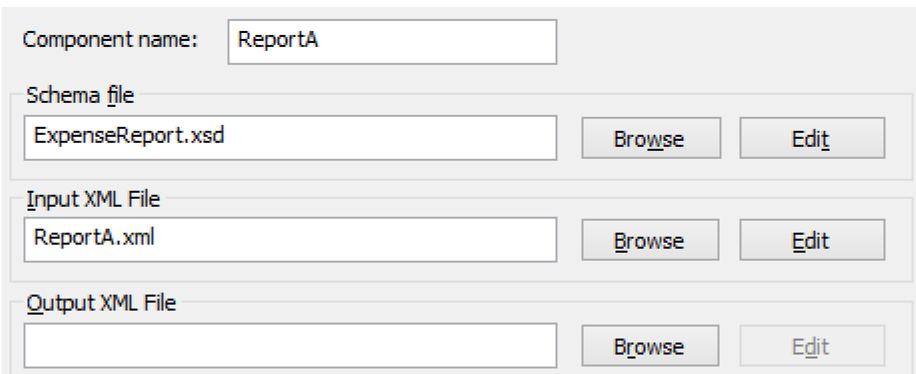
The goal of the mapping it to produce, based on the file above, two further reports:

- **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.

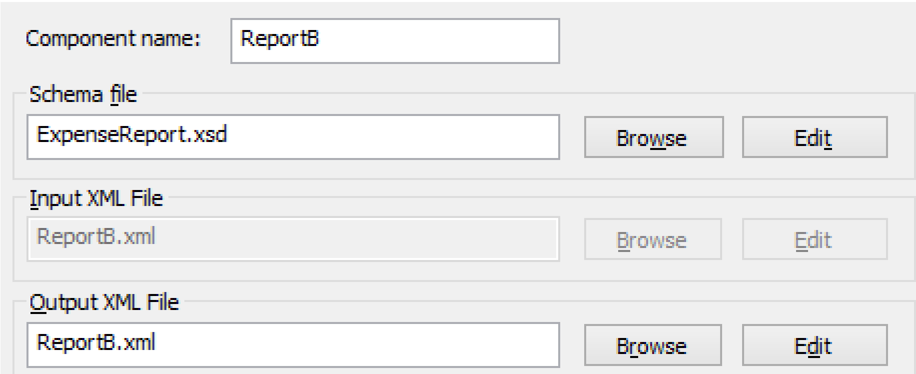


The names of generated output files at each stage in the mapping chain is determined by the settings of each component. (To open the component settings, right-click it, and then select **Properties** from the context menu). Namely, the first component is configured to read data from an XML file called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



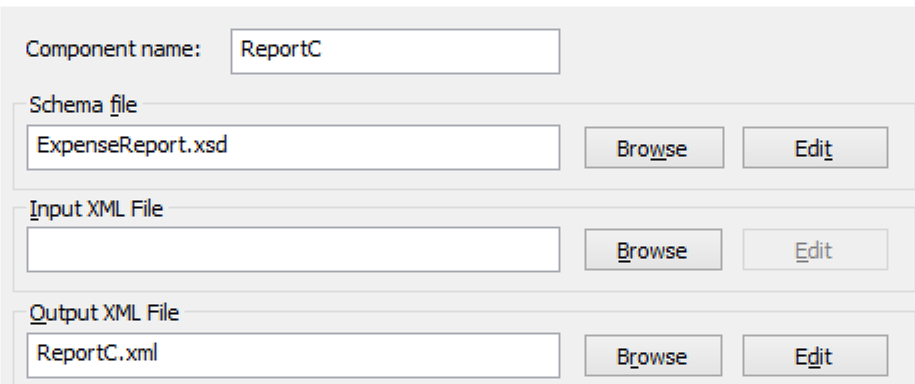
Settings of the source component

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.



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.

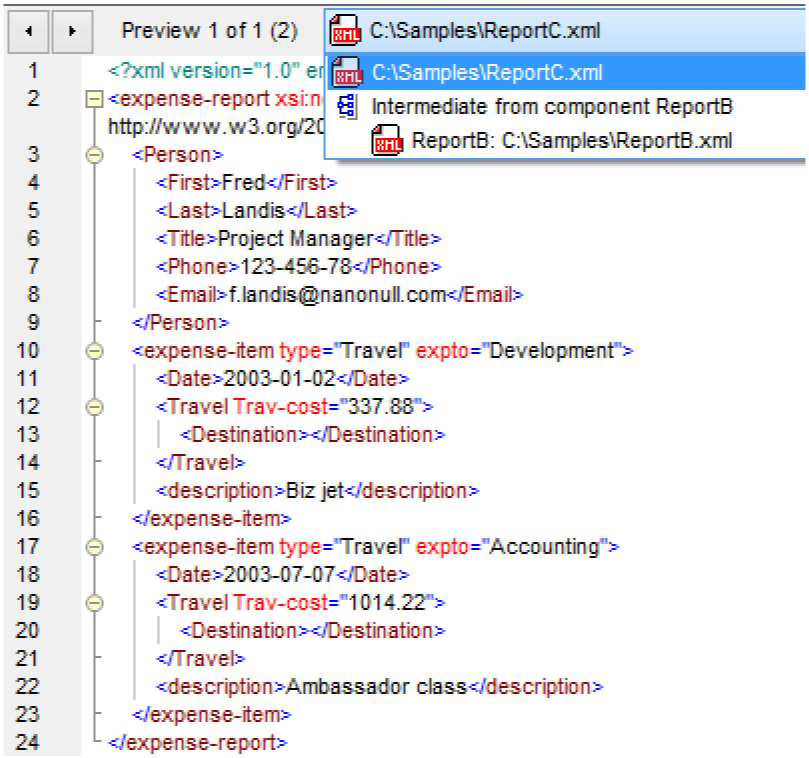


Settings of the target component

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.

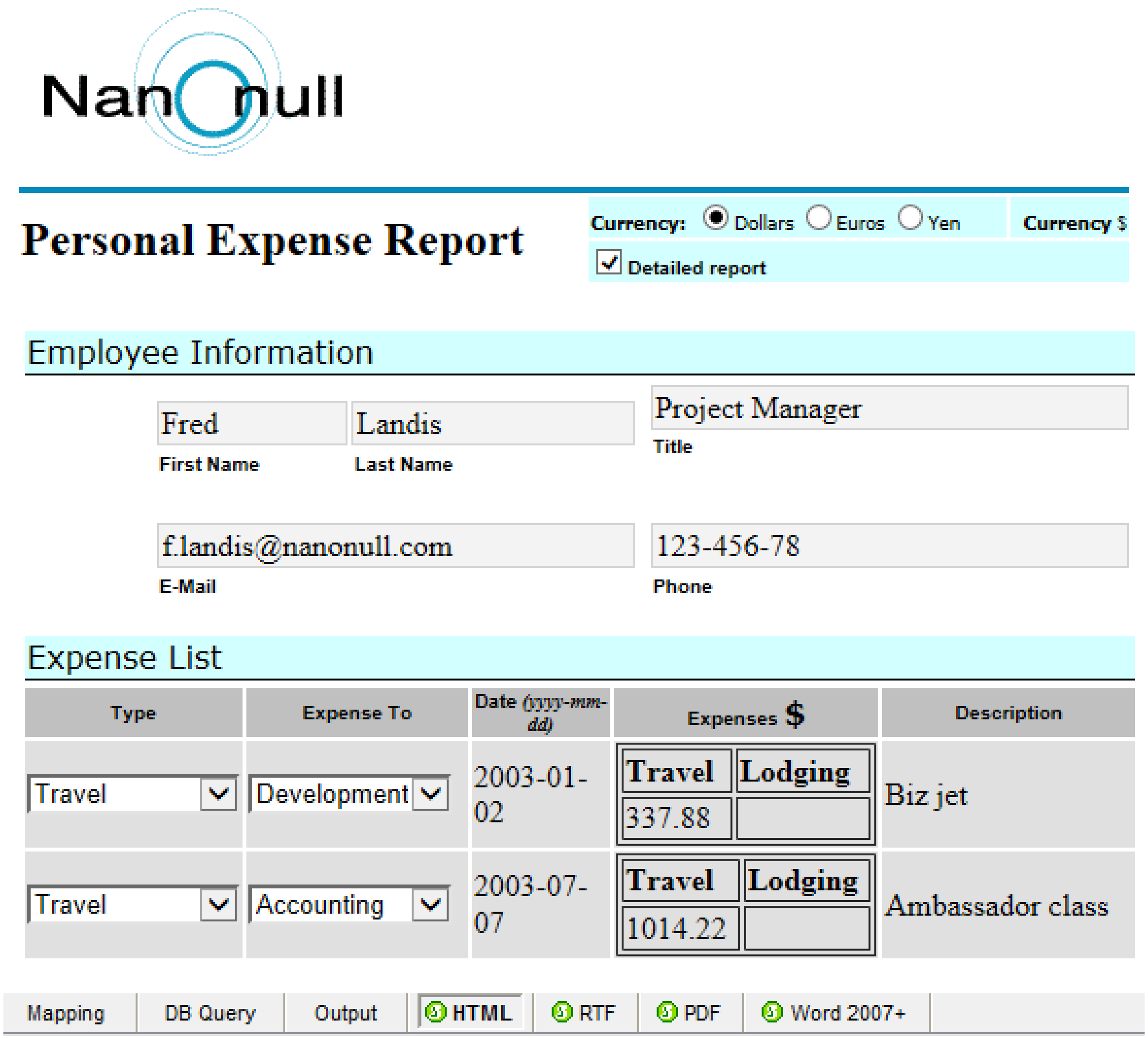
To select which of the two generated output files should be displayed in the Output window, either click the arrow buttons, or select the desired entry from the dropdown list.



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.

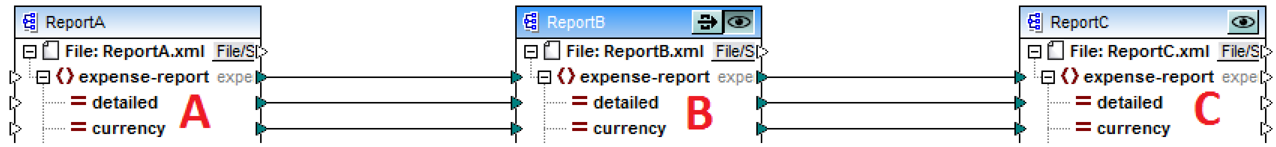


Generated HTML output

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 [377]).

7.1.2 Example: Pass-Through Inactive

The mapping used in this example (**ChainedReports.mfd**) is available in the **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial** folder. This example illustrates how output is generated differently when the pass-through button is deactivated on the intermediate component.

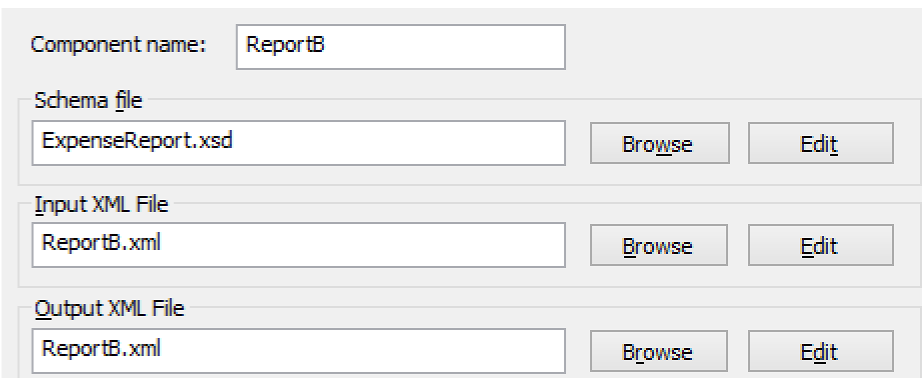


As explained in [Example: Pass-Through Active][373], the goal of the mapping is to produce two separate reports. In the previous example, the pass-through button was active , and both reports were generated as expected and could be viewed in the **Output** tab. However, if you want to preview only one 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

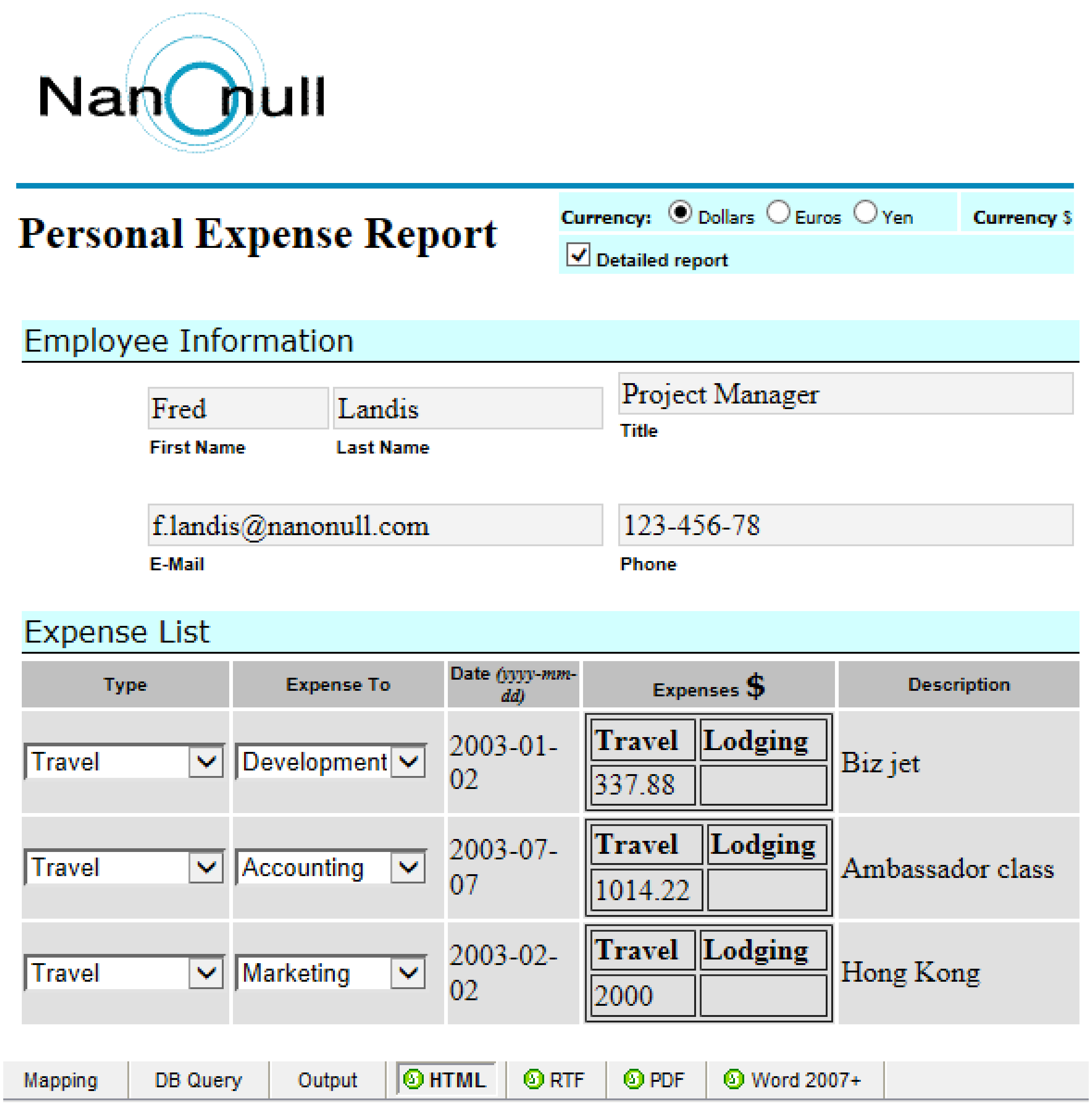
Having the same input and output file on the intermediate component is particularly important if you intend to generate code from the mapping, or run the mapping with MapForce Server. As previously mentioned, in these environments, all outputs created by each component in the mapping chain are generated. 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\MapForce2023\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):



HTML output of the intermediate component

7.2 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, you can perform data conversions such as the one above.

Note: You can also perform the transformation above by using the `node-name` [265] and `static-node-name` [266] 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*, XSLT 2.0, XSLT 3.0, XQuery*, C#*, C++*, Java*.

* These languages require MapForce Professional or Enterprise Edition.

For information about how dynamic node names work, see Getting Access to Node Names 381. For a step-by-step mapping example, see Example: Map Element Names to Attribute Values 392.

7.2.1 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\MapForce2023\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\MapForce2023\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 child elements

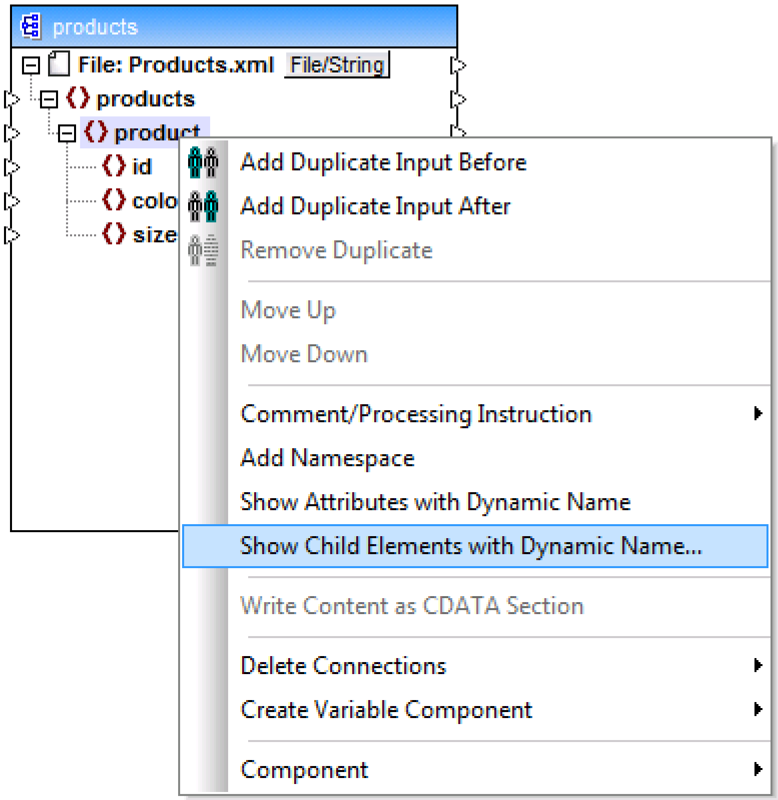


Fig. 1 Enabling dynamic node names (for child elements)

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 389.

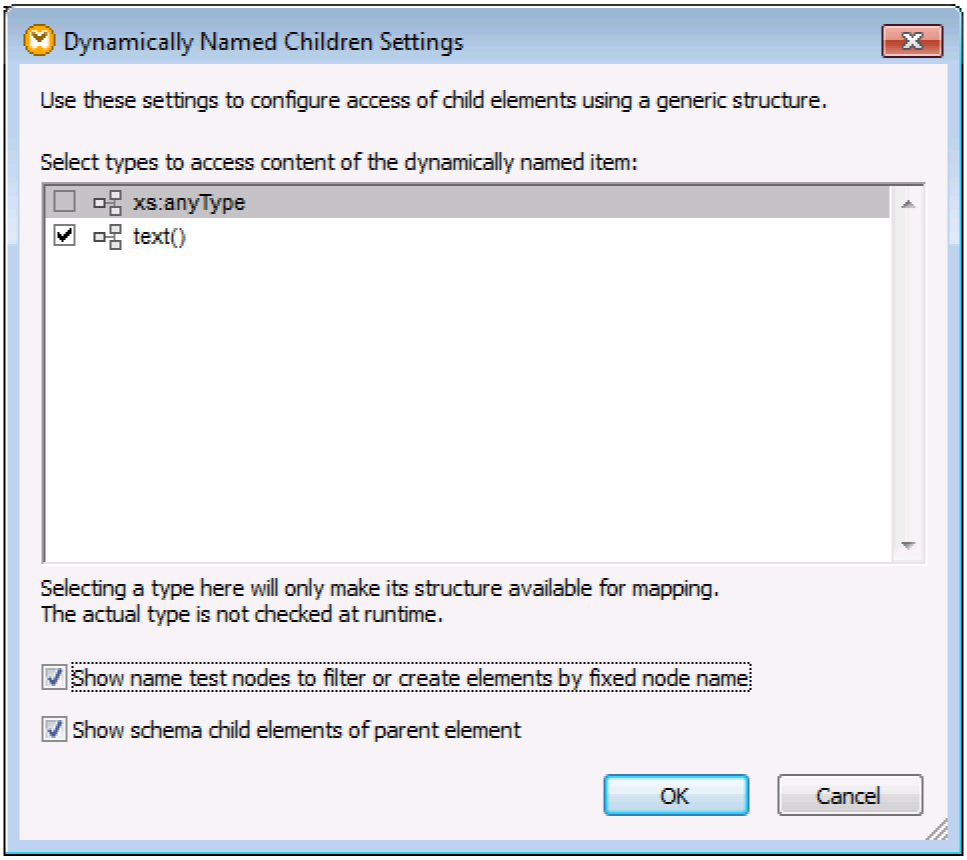


Fig. 2 "Dynamically Named Children Settings" dialog box

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.

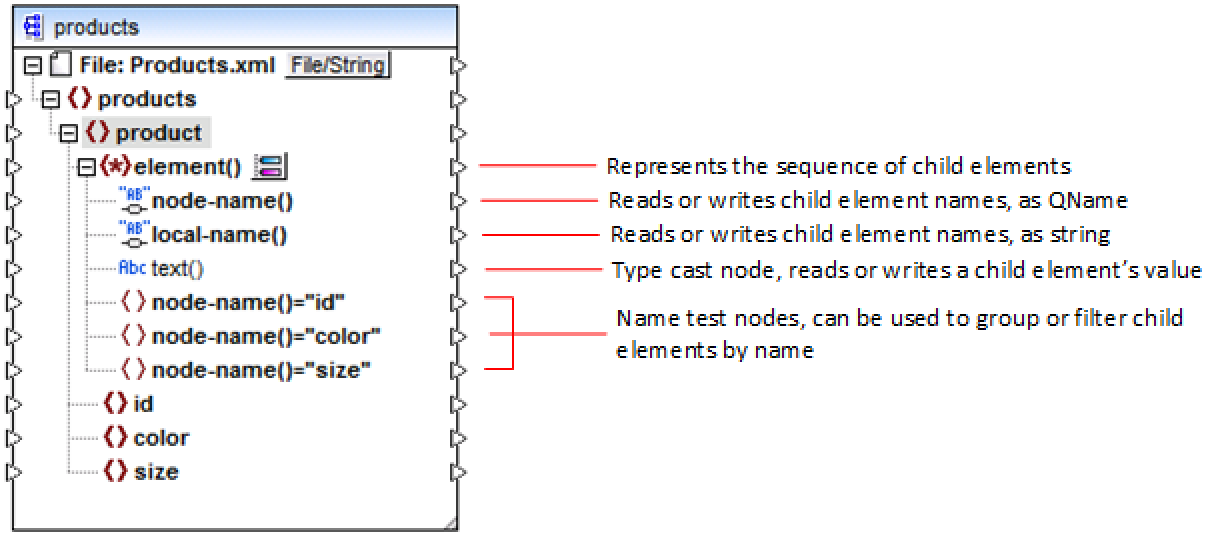


Fig.3 Enabled dynamic node names (for elements)

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.

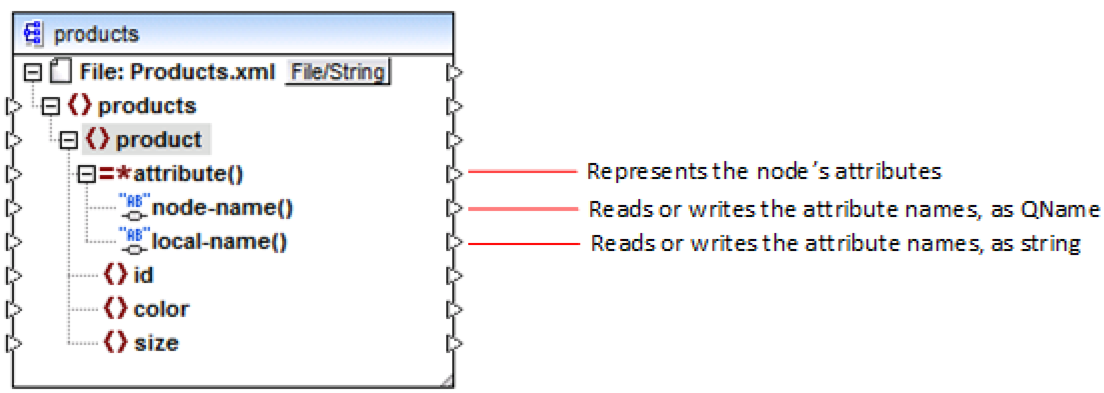


Fig. 4 Enabled dynamic node names (for attributes)

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:

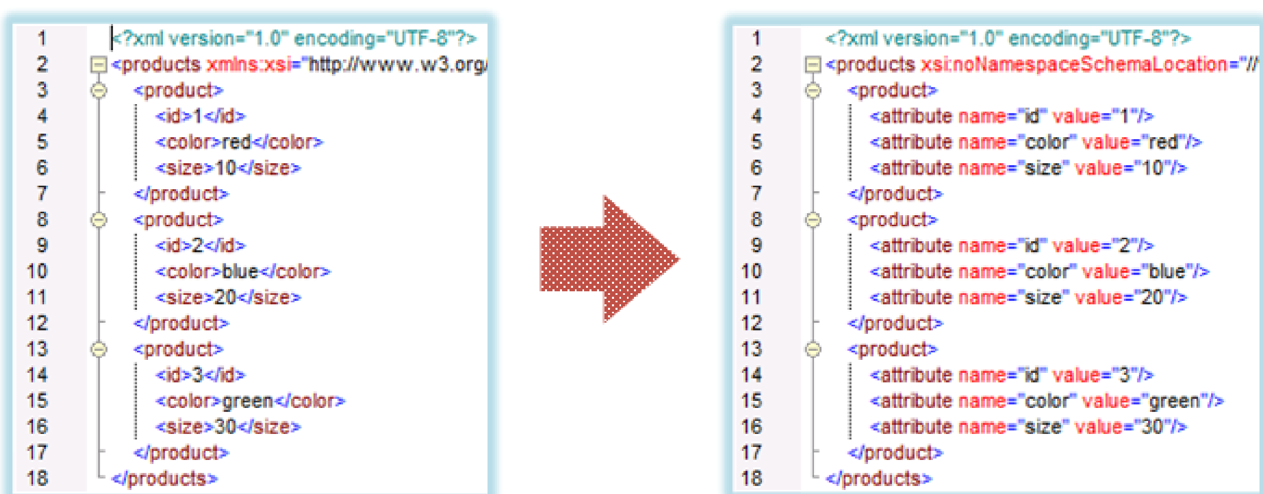


Fig. 6 Mapping XML element names to attribute values (requirement)

The mapping which would achieve this goal looks as follows:

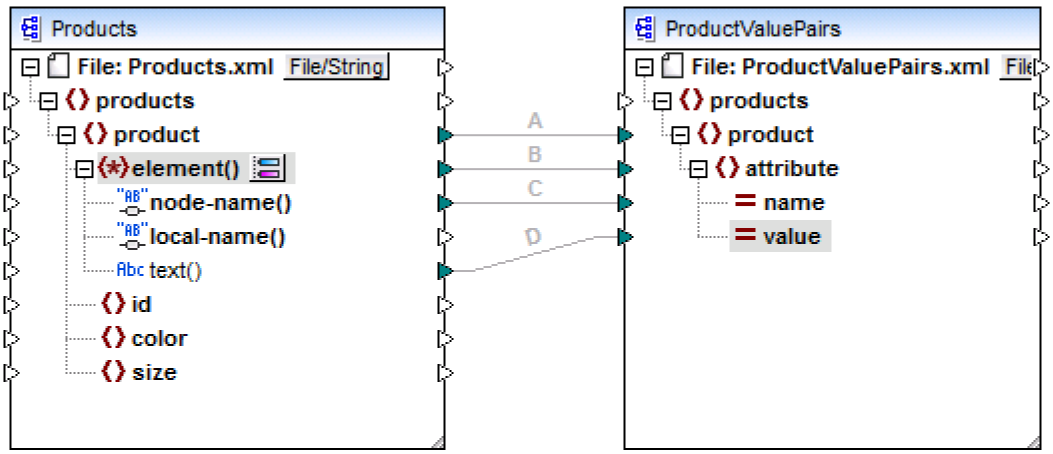


Fig. 7 Mapping XML element names to attribute values (in MapForce)

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 392.

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:

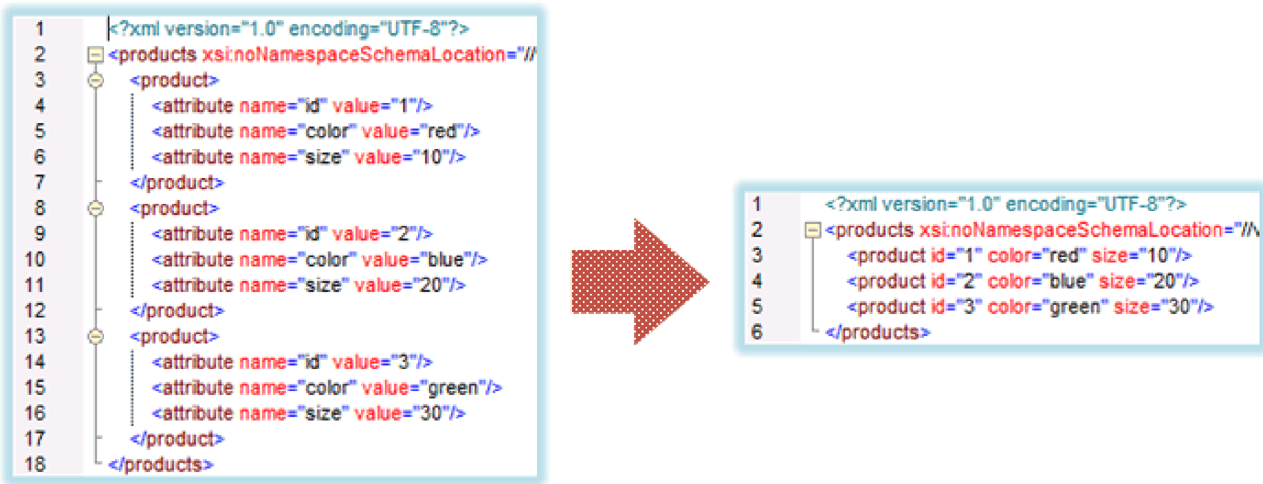


Fig. 8 Mapping attribute values to attribute names (requirement)

The mapping which would achieve this goal looks as follows:

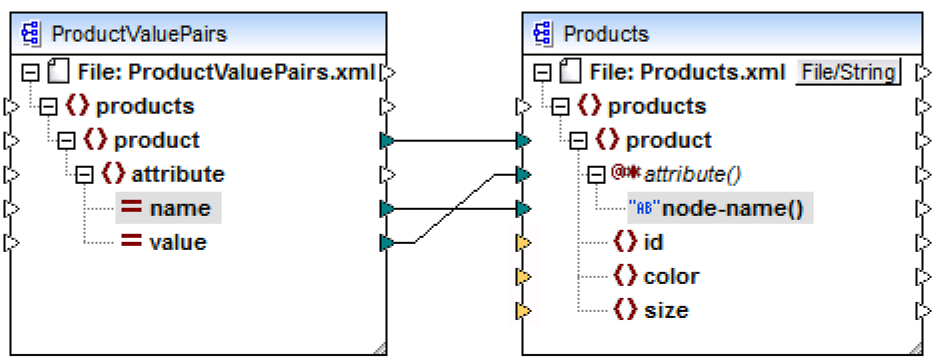


Fig. 9 Mapping attribute values to attribute names (in MapForce)

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\MapForce2023\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 389.

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 389).

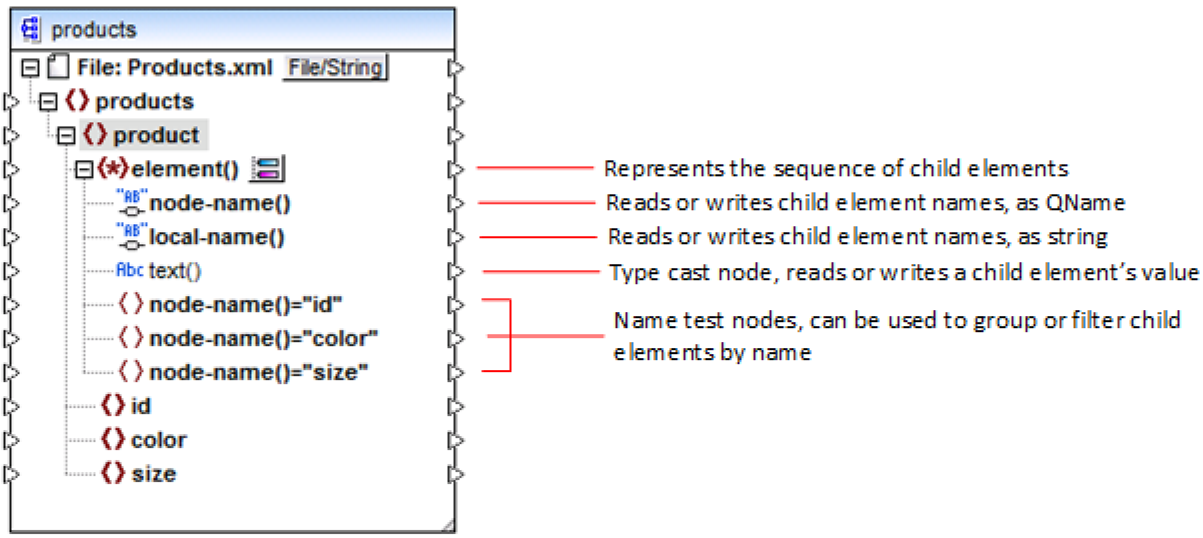
In general, the name test nodes work almost like normal element nodes for reading and writing values and subtree structures. However, because the mapping semantics is different when dynamic access is enabled, there are some limitations. For example, you cannot concatenate the value of two name test nodes.

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()`.

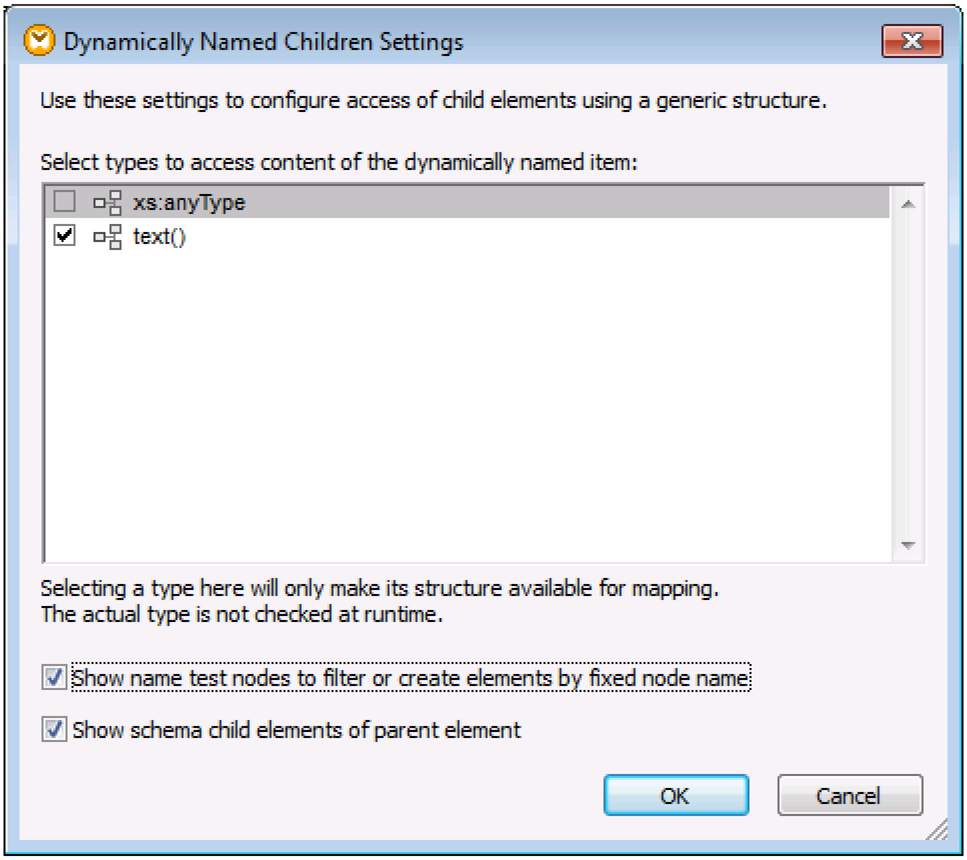
If necessary, you can hide the name test nodes from the component. To do this, click the **Change Selection** ( ) button next to the `element()` node. Then, in the "*Dynamically Named Children Settings*" dialog box (Fig. 2), clear the **Show name test nodes...** check box.

7.2.2 Accessing Nodes of Specific Type

As mentioned in the previous section, Getting Access to Node Names 381, 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 392. 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**.

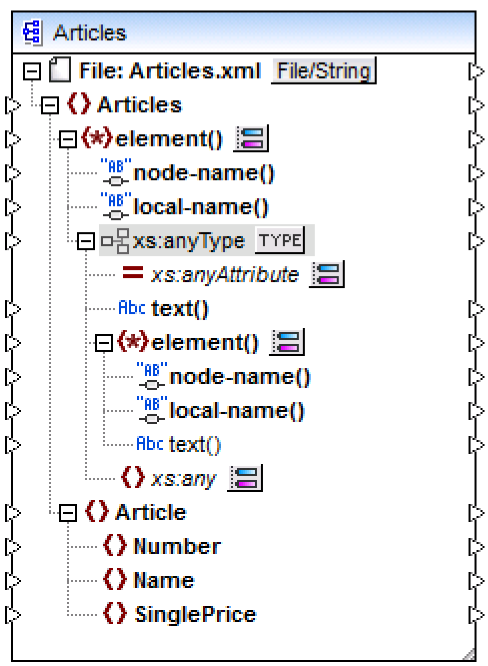
> Be aware that, at mapping runtime, MapForce (through the type cast node) has no information as to what the actual type of the instance node is. Therefore, your mapping must access the node content using the correct type. For example, if you expect that the node of a source XML instance may have children 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 [168].

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 [392], 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", "great-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\MapForce2023\MapForceExamples\Tutorial** 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 [111]).
- 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 [117]). If using this option, make sure that the selected attribute or element can actually exist at that particular level according to the schema.

7.2.3 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\MapForce2023\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:

```xml
<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:

```xml
<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.

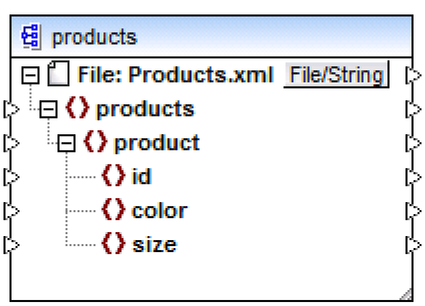
Step 1: Add the source XML component to the mapping

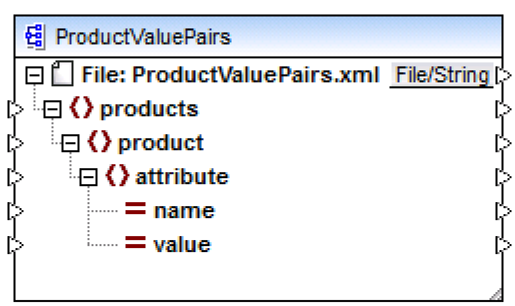
- On the **Insert** menu, click **XML Schema/File**, and browse for the following file: **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\Products.xml**. This XML file points to the **Products.xsd** schema located in the same folder.

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\MapForce2023\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.

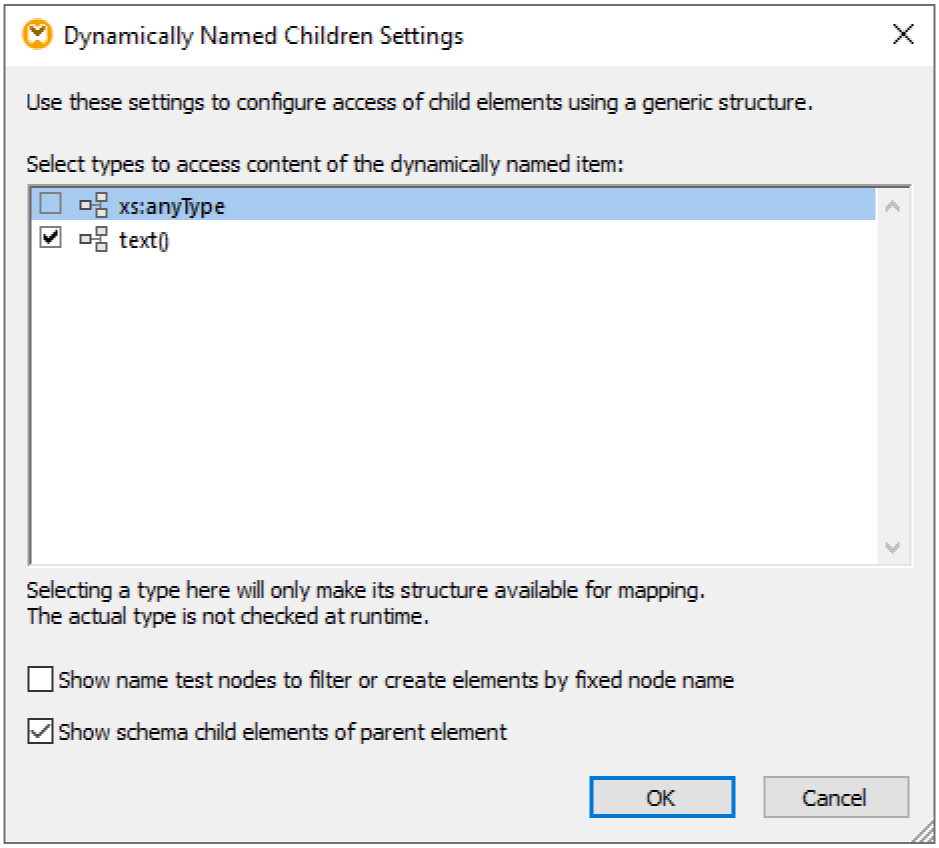
At this stage, the mapping should look as follows:



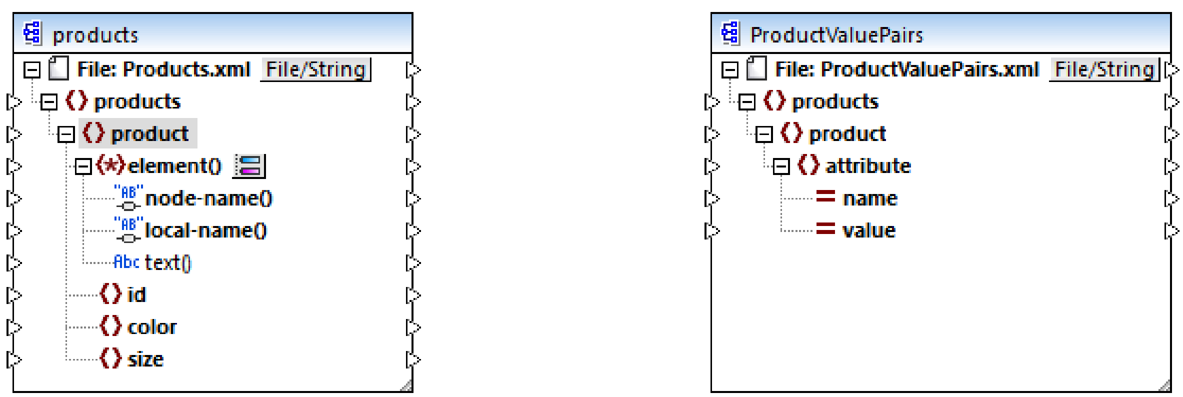


Step 3: Enable dynamic access to child nodes

1. Right-click the `product` 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.

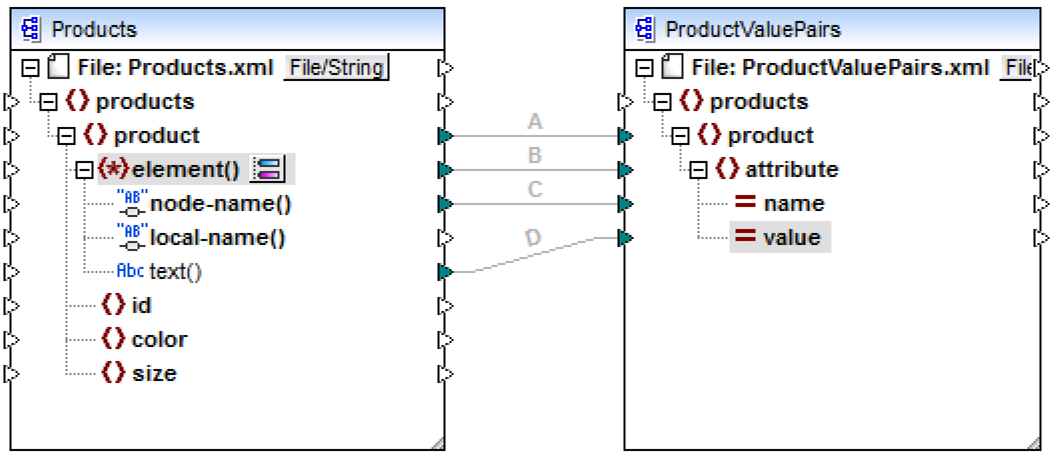


Notice that a `text()` node has been added on the source component. This node will supply the content of each child item to the mapping (in this case, the value of "id", "color", and "size").



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

7.3 Mapping Rules and Strategies

In general, MapForce maps data in an intuitive way, but you may come across situations where the output contains too many or too few items. This chapter is meant to help you avoid situations when the mapping produces undesired output due to incorrect connections or mapping context.

Mapping rules

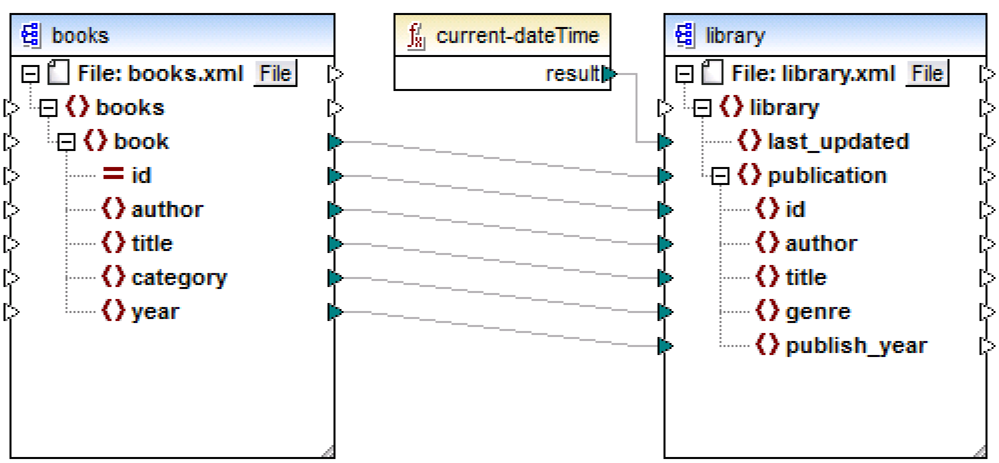
In order to be valid, a mapping must include at least one source and at least one target component. A source component is one that reads data, typically from a file or database. A target component is one that writes data, typically to a file or database. If you attempt to save a mapping where the above is not true, an error appears in the Message window: "A mapping requires at least two connected structures, for a example, a schema or a database structure".

To create a data mapping, you draw mapping connections between items in the source and target components.

All mapping connections that you draw make together a mapping algorithm. At mapping runtime, MapForce evaluates the algorithm and processes data based on it. The integrity and the efficiency of the mapping algorithm depends primarily on the connections. You can also tweak some settings at mapping [103] level, at component [71] level, or even at connection [81] level, but, essentially, the mapping *connections* determine how your data is processed.

Keep in mind the following rules when creating connections:

1. When you draw a connection *from* a source item, the mapping reads data associated with that item from the source file or database. The data may have zero, one, or multiple occurrences (in other words, it may be a sequence, as further described below). For example, if the mapping reads data from an XML file containing books, the source XML file may contain zero, one, or multiple **book** elements. In the mapping below, notice that the **book** item appears only once on the mapping component, even though the source (instance) file may contain multiple **book** elements, or none.



2. When you draw a connection *to* a target item, the mapping generates instance data of that kind. If the source item contains simple content (for example, string or integer) and if the target item accepts simple content, MapForce copies the content to the target item and, if necessary, converts the data type. Zero, one, or multiple values can be generated, depending on the incoming source data, see the next bullet.

3. For each (instance) item in the source, one (instance) item is created in the target. **This is the general mapping rule in MapForce.** Taking the mapping above as example, if the source XML contains three **book** elements, then three **publication** elements will be created on the target side. Note that there are also a few special cases, see Sequences 398.
4. Each connection creates a *current mapping context*. The context determines which data is available *at the current moment, for the current target node*. The context, therefore, determines which source items are actually copied from the source to the target component. By drawing or omitting a connection, you may inadvertently change the current context and thus affect the output of the mapping. For example, your mapping might unnecessarily call a database or a Web service multiple times in the same mapping execution. This concept is further described below, see The mapping context 399.

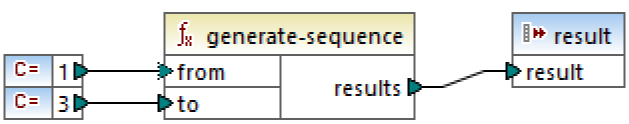
7.3.1 Sequences

As mentioned before, the general mapping rule is "for each item in the source, create one in the target". Here, "item" means one of the following:

- a single instance node of the input file or database
- a sequence of zero to multiple instance nodes of the input file or database

During mapping execution, if a sequence reaches a target item, this creates a loop that generates as many target nodes as there are source nodes. There are some exceptions to this rule, however:

- If the target item is an XML root element, it is created once and only once. If you connect a sequence to it, the result might not be schema valid. If attributes of the root element are also connected, the XML serialization will fail at mapping runtime. Therefore, avoid connecting a sequence to the root XML element.
- If the target item accepts only one value, it is created only once. Examples of items that accept only one value: XML attributes, database fields, simple output components. For example, the mapping below generates a sequence of three integers (1, 2, 3) with the help of the **`generate-sequence`** function. Nevertheless, the output will contain only one integer, because the target is a simple output component that accepts a single value. The other two values are ignored.



- If the source schema specifies that a specific item occurs only once, but the instance file contains many, MapForce may extract the first item from the source (which must exist according to the schema) and create only one item in the target. To disable this behavior, clear the check box **Enable input processing optimizations based on min/maxOccurs** from the component settings, see also XML Component Settings 107.

If the sequence is empty, nothing is generated on the target side. For example, if the target is an XML document and the source sequence is empty, no XML elements would be created in the target at all.

Functions work in a similar way: if they get a sequence as input, then they are called as many times as (and produce as many results as) there are items in the sequence.

If a function gets an empty sequence as input, it returns an empty result as well, and consequently

produces no output at all.

However, there are some categories of functions that, by virtue of their design, return a value even if they get an empty sequence as input:

- **exists**, **not-exists**, **substitute-missing**
- **is-null**, **is-not-null**, **substitute-null** (these three functions are aliases of the previous three)
- aggregate functions (**sum**, **count**, and so on)
- user-defined functions that accept sequences and are regular (not inlined) functions

If you need to replace an empty value, add the **substitute-missing** function to the mapping and replace the empty value with a substitute value of choice.

Functions may have multiple inputs. If a sequence is connected to each input, this produces a Cartesian product of all inputs, which is typically not the desired outcome. To avoid this, connect only one sequence to a function with multiple parameters; all other parameters must be connected to "singular" items from parents or other components.

7.3.2 The Mapping Context

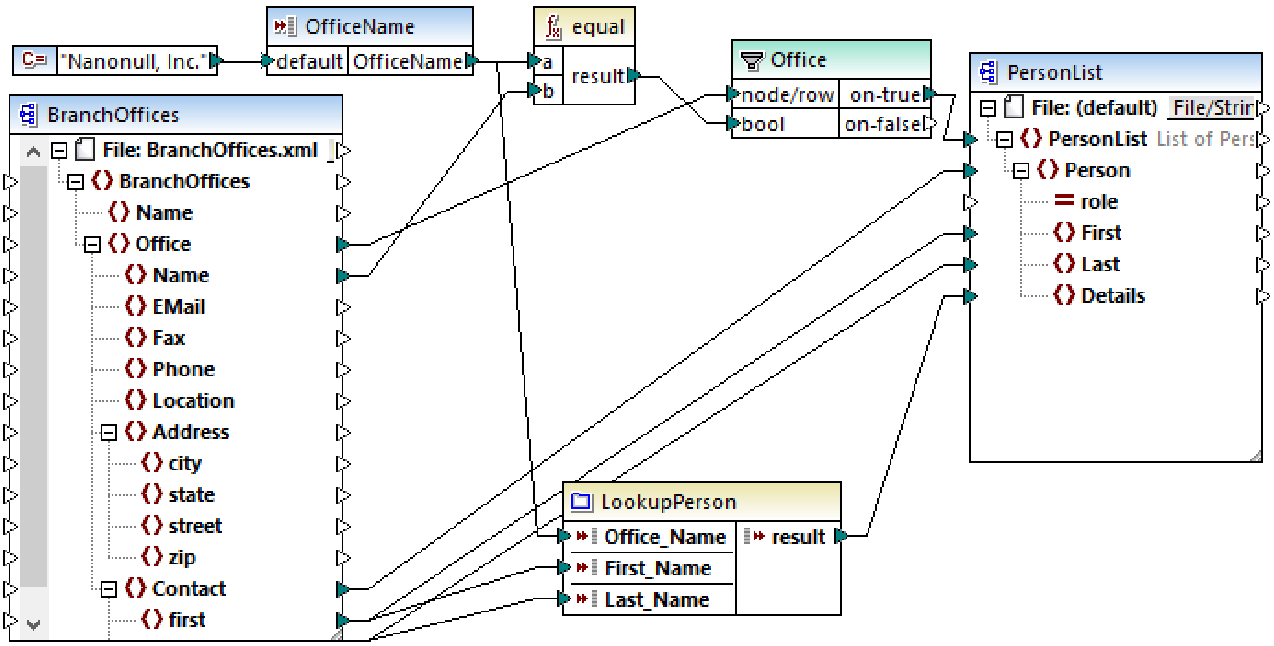
Mapping components are hierarchical structures that may contain many levels of depth. On the other hand, a mapping may have multiple source and components, plus any intermediary components such as functions, filters, value-maps, and so on. This adds complexity to the mapping algorithm, especially when multiple unrelated components are connected. To make it possible to execute the mapping in portions, one step at a time, a current context must be established for each connection.

We could also say that multiple "current contexts" are established for the duration of the mapping execution, since the current context changes with each processed connection.

MapForce always establishes the current context starting from the *target root item (node)*. This is where the mapping execution actually begins. The connection to the target root item is traced back to all source items that are directly or indirectly connected to it, including via functions or other intermediary components. All the source items and results produced by functions are added to the current context.

After it finishes processing the target node, MapForce works down the hierarchy. Namely, it processes all *mapped items* of the target component from top to bottom. For each new item, a new context is established that initially contains all items of the parent context. Thus, all mapped sibling items in a target component are independent of each other, but have access to all source data of their parent items.

Let's see how the above applies in practice, based on an example mapping, **PersonListByBranchOffice.mfd**. You can find this mapping in the **<Documents>\Altova\MapForce2023\MapForceExamples** directory.



In the mapping above, both the source and the target component are XML. The source XML file contains two **Office** elements.

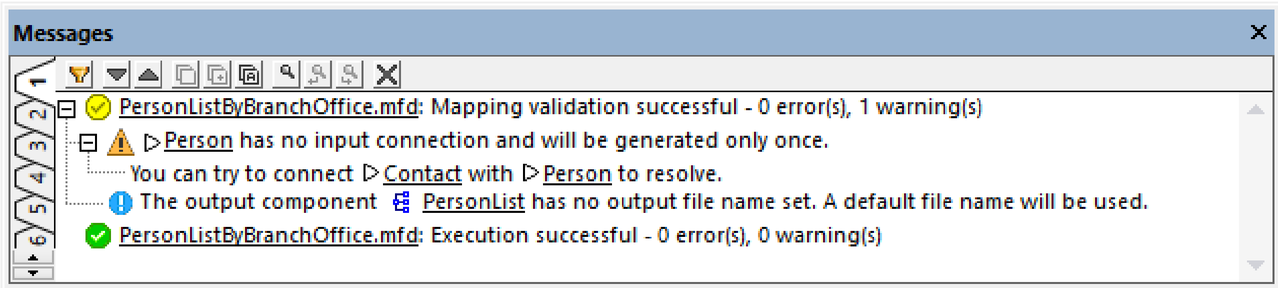
As mentioned previously, the mapping execution always begins from the target root node (**PersonList**, in this example). By tracing back the connection (via the filter and the function) to a source item, you can conclude that the source item is **Office**. (The other connection path leads to an input parameter and its purpose is further explained below).

Had there been a straightforward connection between **Office** and **PersonList**, then, according to the general mapping rule, the mapping would have created as many **PersonList** instance items as there are **Office** items in the source file. However, this does not happen here, because there is a filter in between. The filter supplies to the target component only data that satisfies the Boolean condition connected to the **bool** input of the filter. The **`equal`** function returns **true** if the office name is equal to "Nanonull, Inc.". This condition is satisfied only once, because there is only one such office name in the source XML file.

Consequently, the connection between **Office** and **PersonList** defines a single office as the context for the entire target document. This means that all descendants of the **PersonList** item have access to data of the office "Nanonull, Inc." office, and no other office exists in the current context.

The next connection is between **Contact** and **Person**. According to the general mapping rule, it will create one target **Person** for each source **Contact**. On each iteration, this connection establishes a new current context; therefore, the child connections (**first** to **First**, **last** to **Last**) supply data from the source to the target item in the context of each **Person**.

If you left out the connection between **Contact** and **Person**, then the mapping would create only one **Person** with multiple **First**, **Last**, and **Details** nodes. In such cases, MapForce issues a warning and a suggestion in the Messages window, for example:



Finally, the mapping includes a user-defined function, **LookupPerson**. The user-defined function is also executed in the context of each **Person**, because of the parent connection between **Contact** and **Person**. Specifically, each time when a new Person item is created on the target side, the function is called to populate the **Details** element of the person. This function takes three input parameters. The first one (**OfficeName**) is set to read data from the input parameter of the mapping. The source data for this parameter could as well be provided by the **Name** source item, without changing in any way the mapping output. In either case, the source value is the same and it is taken from a parent context. Internally, the look-up function concatenates the values received as arguments and produces a single value. For more information about how the **LookupPerson** function works, see the Example: Look-up and Concatenation 210.

7.3.2.1 User-Defined Functions

User-defined functions (UDFs) are custom functions embedded into the mapping, where you define the inputs, outputs, and processing logic. Each user-defined function may contain the same component kinds as a main mapping, including Web services and databases.

> By default, if a UDF contains a database or a Web service component, and if the input data to the UDF is a sequence of multiple values, then each input value will call the UDF and consequently will result in a database or Web service call.

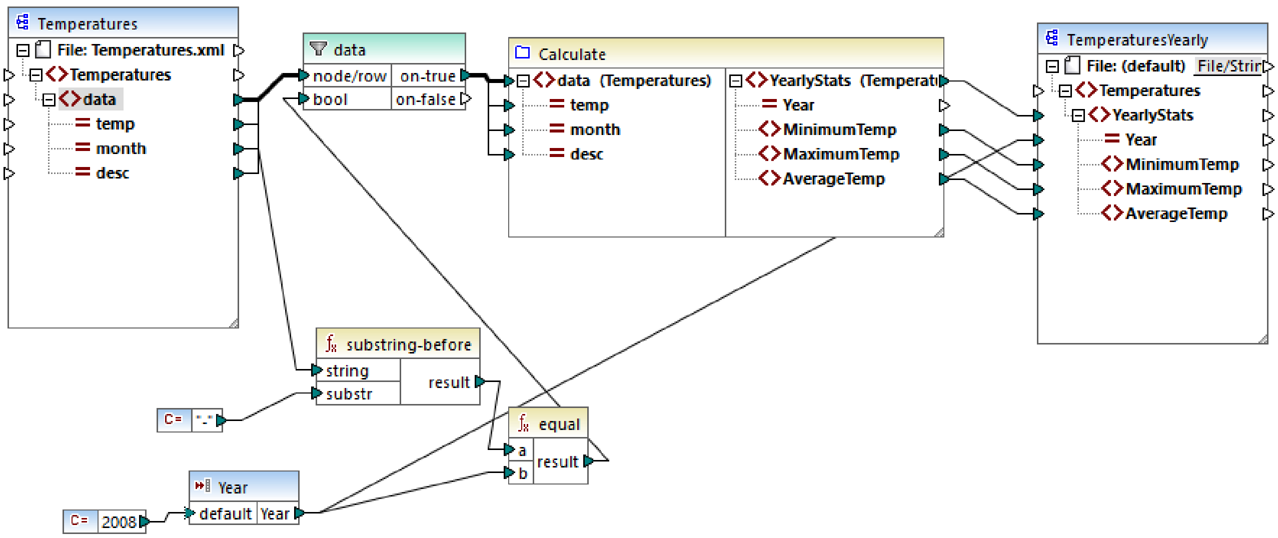
The behavior above may be acceptable for those mappings where you really need the UDF to be called *as many times as there are input values* and there is simply no other alternative way.

If you do not want the above to happen, you can configure the UDF so that it is called only once even if gets a sequence of values as input. You will typically want to do this for those UDFs that operate on a set of values before they can return (such as functions that calculate averages or totals).

Configuring a UDF to accept multiple input values in the same function call is possible if the UDF is of type "regular", not "inlined". (For details, see the User-Defined Functions 198 chapter.) With regular functions, you can specify that the input parameter is a sequence by selecting the **Input is a sequence** check box. This check box is visible on the component settings, after you double-click the title bar of an input parameter. The check box affects how often the function is called, as follows:

- When input data is connected to a **sequence** parameter, the user-defined function is called *only once* and the complete sequence is passed into the user-defined function.
- When input data is connected to a **non-sequence** parameter, the user-defined function is called *once for each single item in the sequence*.

For an example, open the following demo mapping:
<Documents>\Altova\MapForce2023\MapForceExamples\InputIsSequence.mfd.

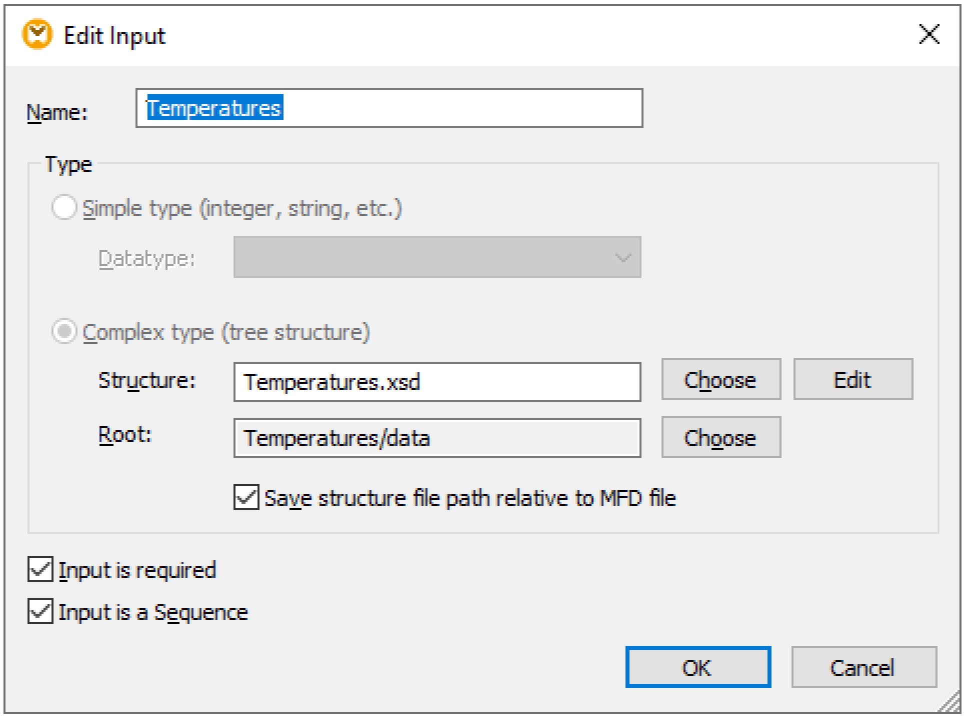


The mapping above illustrates a typical case of a UDF that operates on a set of values and thus requires all the input values in one call. Specifically, the `Calculate` user-defined function returns the minimum, maximum and average temperatures, taking as input data from a source XML file. The expected mapping output is as follows:

```
<Temperatures>
    <YearlyStats Year="2008">
        <MinimumTemp>-0.5</MinimumTemp>
        <MaximumTemp>24</MaximumTemp>
        <AverageTemp>11.6</AverageTemp>
    </YearlyStats>
</Temperatures>
```

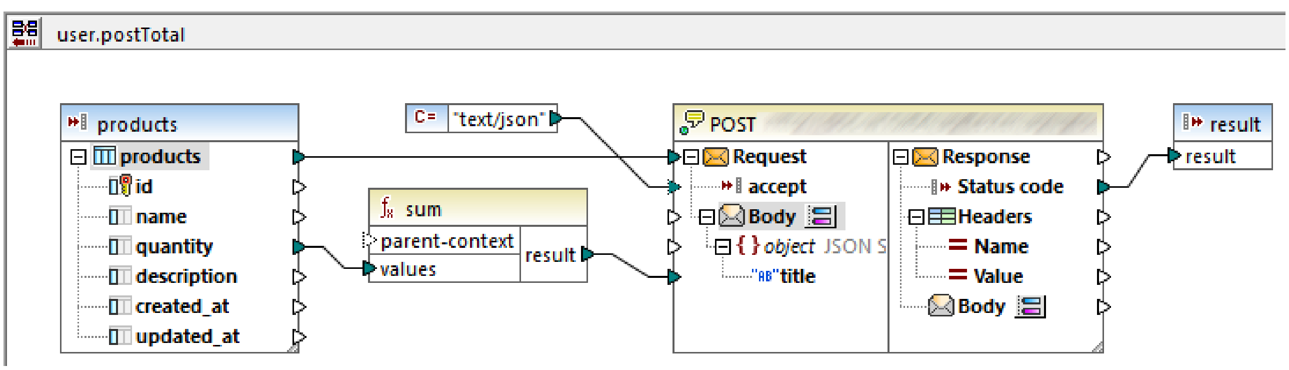
As usual, the mapping execution begins with the top item of the target component (**YearlyStats**, in this example). To populate this node, the mapping attempts to obtain source data from the UDF, which in its turn, triggers the filter. The filter's role in this mapping is to pass onto the UDF only temperatures from year 2008.

The check box **Input is sequence** was selected for the input parameter of the UDF (To view this check box, double-click the title bar of the `Calculate` function to enter the function's mapping; then double-click the title bar of the input parameter). As mentioned before, the **Input is sequence** option causes the complete sequence of values to be supplied as input to the function and the function is called only once.



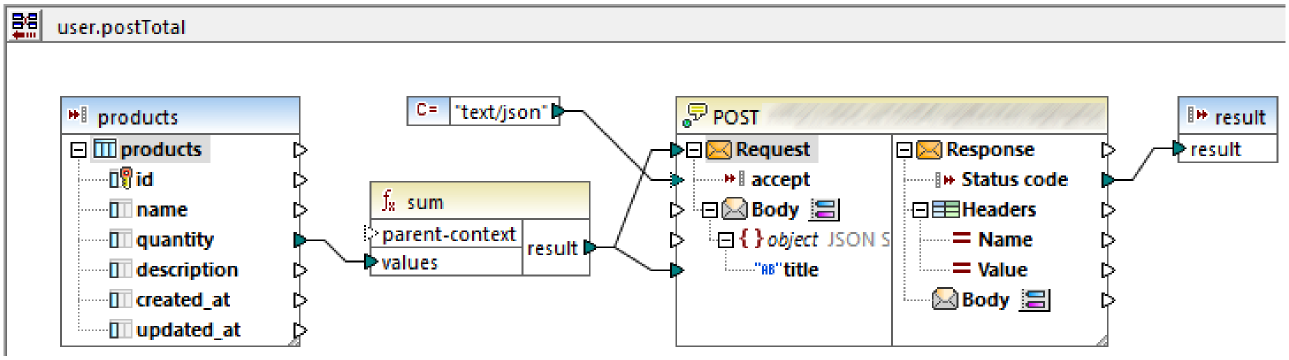
Had the **Input is sequence** check box not been selected, the UDF would have been called for each value in the source. As a result, the minimum, maximum and average would be calculated for each single value individually and incorrect output would be produced.

By applying the same logic in more complex UDFs that include database or Web service calls, it may be possible to optimize the execution and avoid unnecessary calls to the database or Web service. Nevertheless, be aware that the **Input is sequence** check box does not control what happens to the sequence of values *after* it enters the function. In other words, there is nothing to prevent you from connecting the incoming sequence of values to the input of a Web service and thus call it multiple times. Consider the following example:



The UDF illustrated above receives a sequence of values from the external mapping. Specifically, the data supplied to the input parameter originates from a database. The input parameter has the option **Input is sequence** selected, so the entire sequence is supplied to the function in one call. The function is supposed to add up multiple **quantity** values and post the result to a Web service. Exactly one Web service call is expected. However, the Web service will be incorrectly called multiple times when the mapping runs. The reason is that the **Request** input of the Web service receives *a sequence of values*, not a single value.

To fix this problem, connect the **Request** input of the Web service to the result of the **sum** function. The function produces one single value, so the Web service will also be called once:



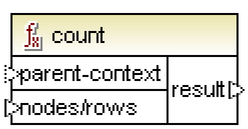
> Normally, aggregate functions like **sum**, **count**, etc produce a single value. Nevertheless, if there is a parent connection that allows it, they may produce a sequence of values as well, as described further in the Example: Changing the Parent Context 404.

7.3.2.2 Example: Changing the Parent Context

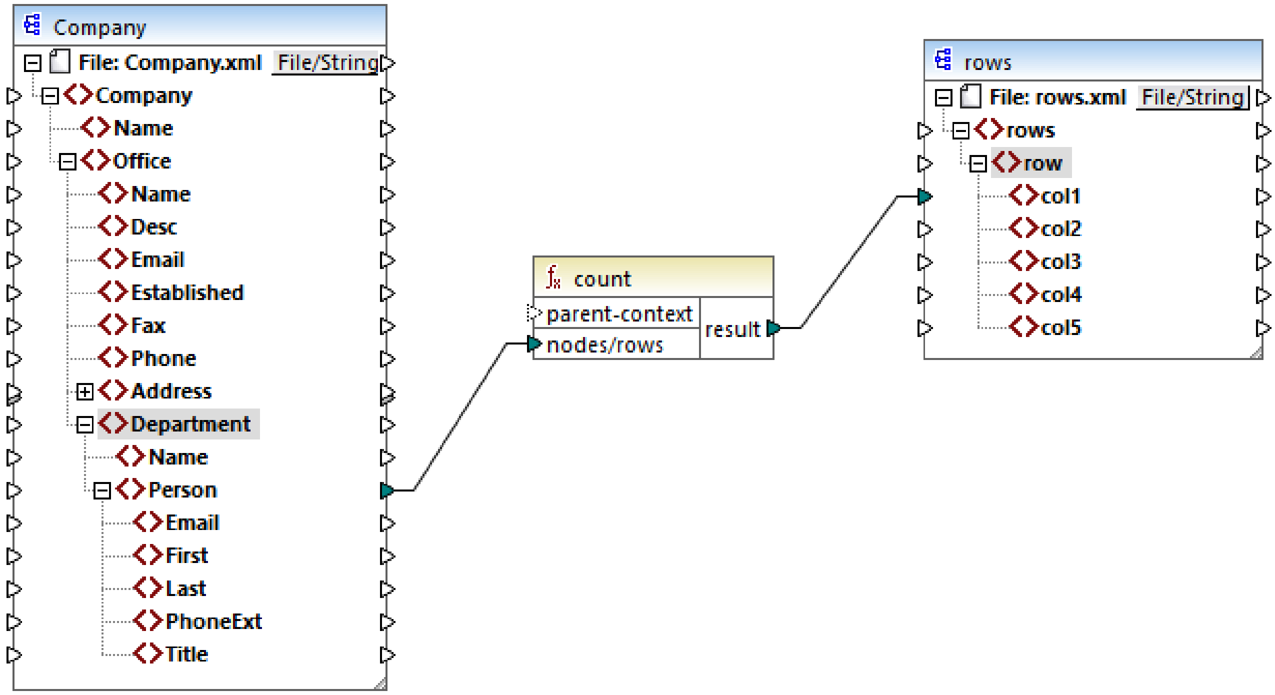
Some mapping components have an optional **parent-context** item.

The `parent-context` argument is an optional argument in some MapForce core aggregation functions (e.g., **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.

With the help of this item you can influence the mapping context in which that component should operate and consequently change the mapping output. The components that have an optional **parent-context** are: aggregate functions, variables, and Join components.



For a demo of how changing the parent context is useful, open the following mapping:
<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\ParentContext.mfd.



In the source XML of the mapping above, there is a single **Company** node which contains two **Office** nodes. Each **Office** node contains multiple **Department** nodes, and each **Department** contains multiple **Person** nodes. If you open the 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

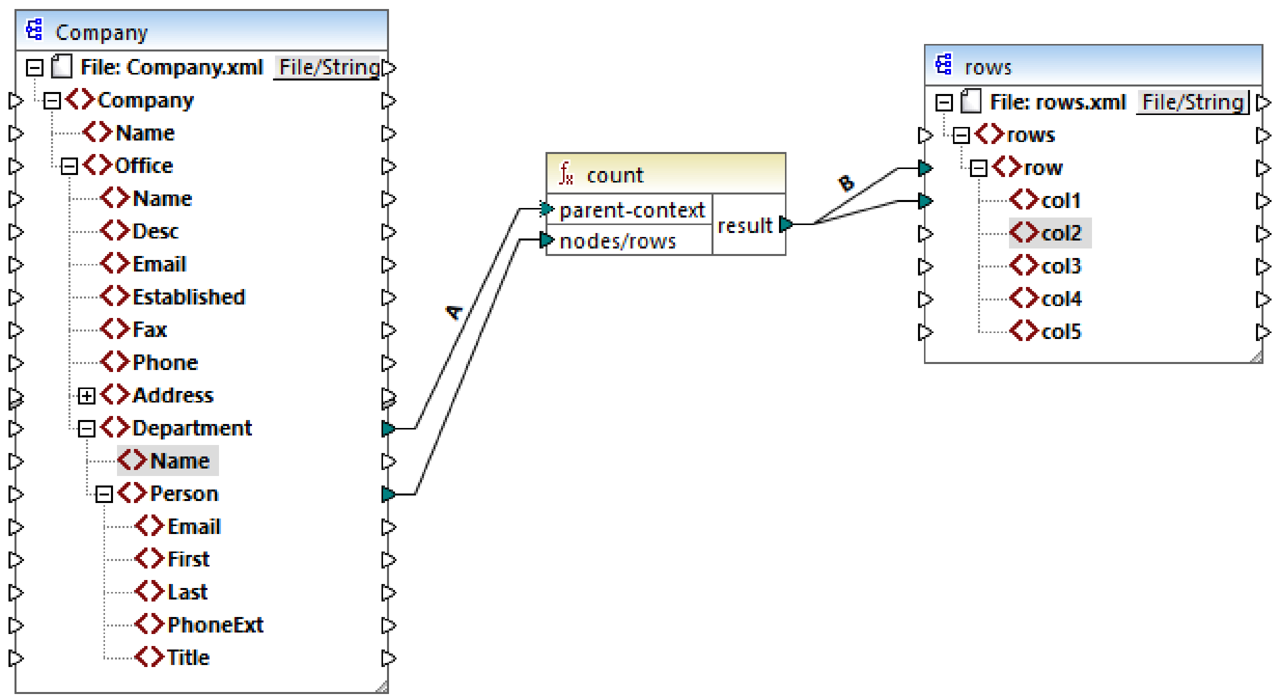
The mapping counts all people in all departments. For this purpose, it uses the **count** function from the `core` library. If you click the **Output** tab to preview the mapping, you will notice that it produces a single value, **21**, which corresponds to the total number of people in the source XML file.

The mapping works as follows:

- As usual, the mapping execution starts from the top node of the target component (**rows**, in this example). There is no incoming connection to **rows**. As a result, an implicit mapping context is established between **Company** (top item of the source component) and **rows** (top item of the target component).

- The function's result is a single value, because there is only one company in the source file.
- To populate the **col1** target item, MapForce executes the **count** function in the *implicit parent context* mentioned above, so it will count all **Person** nodes from all offices and from all departments.

The **parent-context** argument of the function lets you change the mapping context. This enables you, for example, to count the number of people in each department. To do this, draw two more connections as shown below:

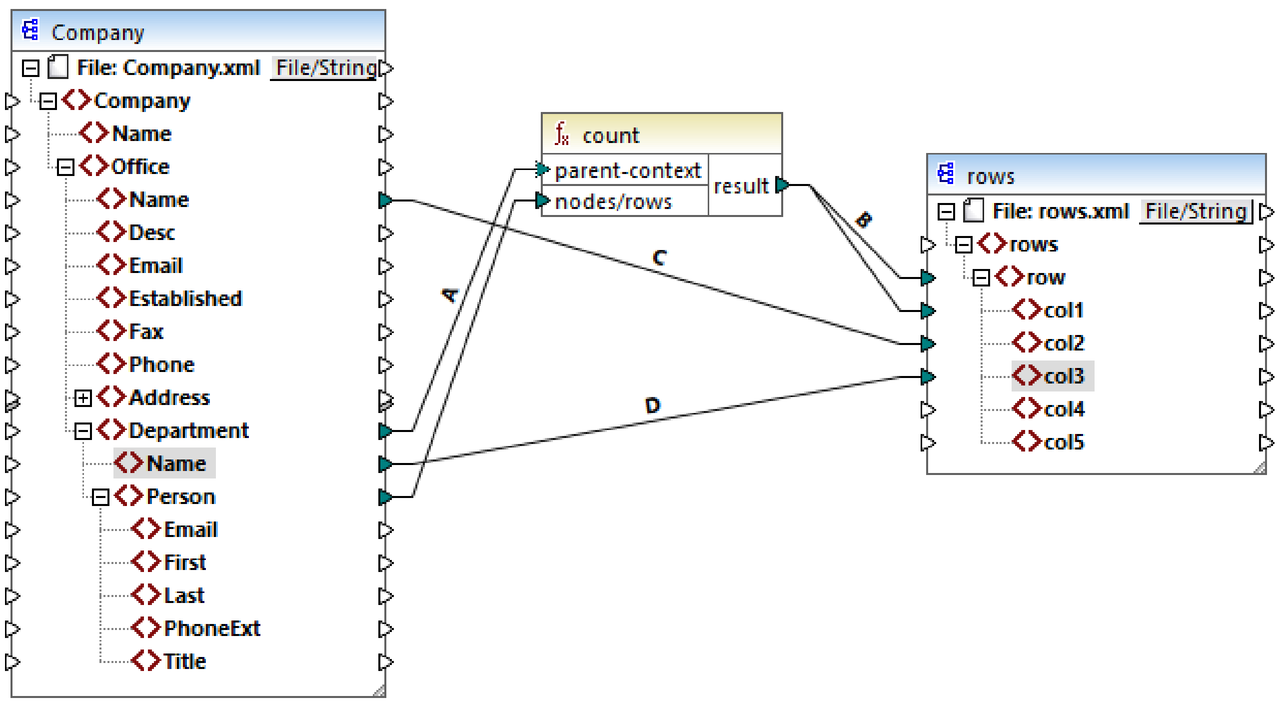


In the mapping above, connection A changes the parent context of the **count** function to **Department**. As a result, the function will count the number of people in each department. Very importantly, the function will now return a *sequence* of results instead of a single result, because multiple departments exist in the source. This is the reason why connection B exists: for each item in the resulting sequence it creates a new row in the target file. The mapping output has now changed accordingly (notice the numbers correspond exactly to the count of people in each department):

```
<rows>
    <row>
        <col1>3</col1>
    </row>
    <row>
        <col1>2</col1>
    </row>
    <row>
        <col1>6</col1>
    </row>
    <row>
```

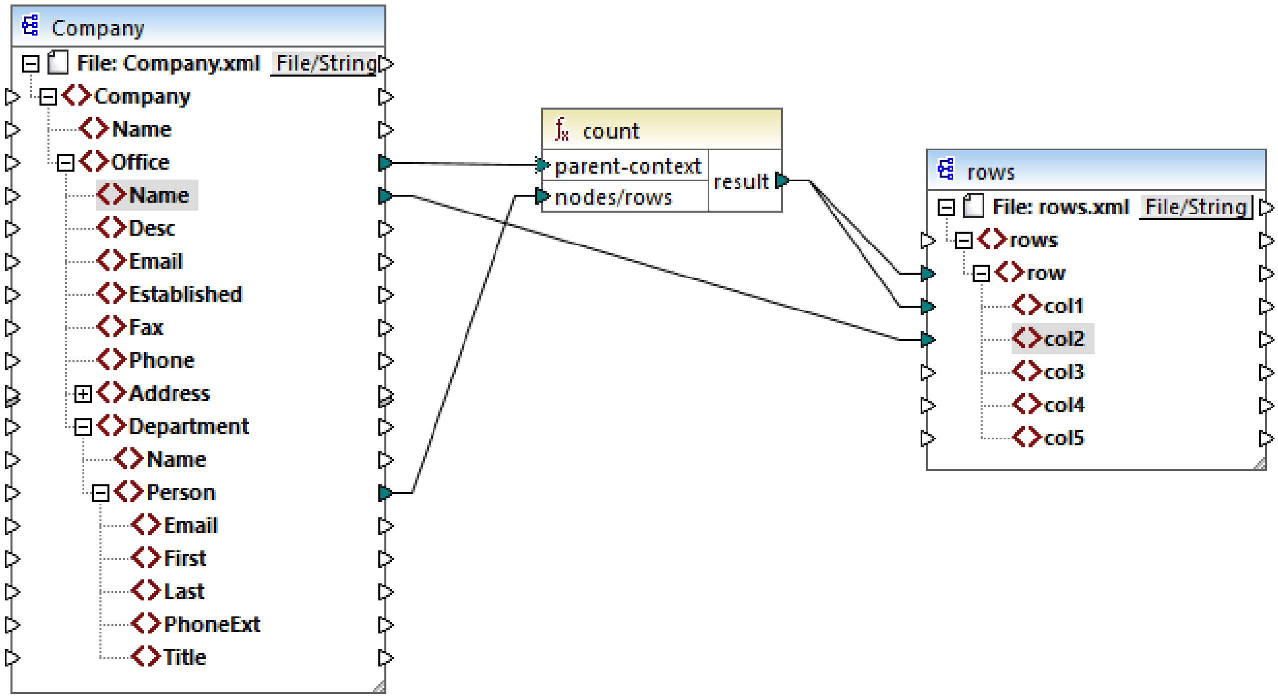
```
        <col1>4</col1>
    </row>
    <row>
        <col1>2</col1>
    </row>
    <row>
        <col1>1</col1>
    </row>
    <row>
        <col1>3</col1>
    </row>
</rows>
```

Given that the current mapping creates a row for each department, you can optionally copy the office name and the department name as well into the target file, by drawing connections C and D:



This way, the output will display not only the count of people but also the corresponding office and department name.

If you would like to count the number of people in each office, connect the parent context of **count** function to the **Office** item in the source.



With the connections shown above, the **count** function returns one result for each office. There are two offices in the source file, so the function will now return two sequences. Consequently, there will be two rows in the output, where each row is the number of people in that office:

```
<rows>
    <row>
        <col1>15</col1>
        <col2>Nanonull, Inc.</col2>
    </row>
    <row>
        <col1>6</col1>
        <col2>Nanonull Partners, Inc.</col2>
    </row>
</rows>
```

7.3.3 Priority context

Priority context is a way to influence the order in which input parameters of a function are evaluated. Setting a priority context may be necessary if your mapping joins data from two unrelated sources.

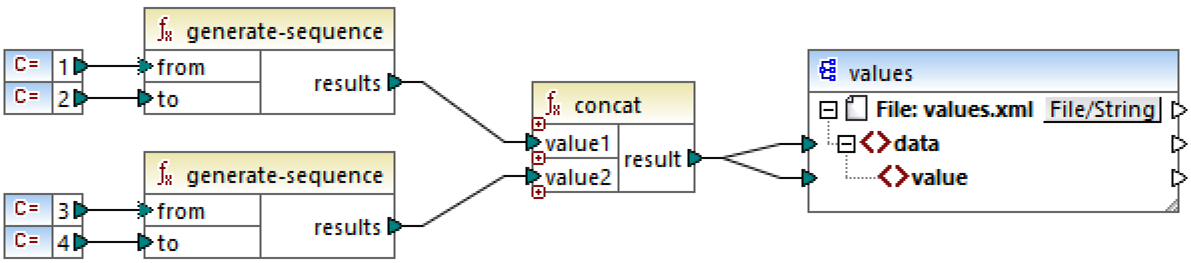
To understand how priority context works, recall that, when a mapping runs, the connection to an input item may carry a *sequence* of multiple values. For functions with two input parameters, this means that MapForce must create two loops, one of which must be processed first. The loop that is processed first is the "outer" loop. For example, the **equal** function receives two parameters: *a* and *b*. If both *a* and *b* get a sequence of values, then MapForce processes as follows:

- For each occurrence of *a*
  - For each occurrence of *b*
    - Is *a* equal to *b*?

As you can see from above, each *b* is evaluated in the context of each *a*. Priority context lets you alter the processing logic so that each *a* is evaluated in the context of each *b*. In other words, it lets you swap the inner loop with the outer loop, for example:

- For each occurrence of *b*
  - For each occurrence of *a*
    - Is *a* equal to *b*?

Let's now examine a mapping where priority context affects the mapping output. In the mapping below, the **concat** function has two input parameters. Each input parameter is a sequence that was generated with the help of the **generate-sequence** function. The first sequence is "1,2" and the second sequence is "3,4".



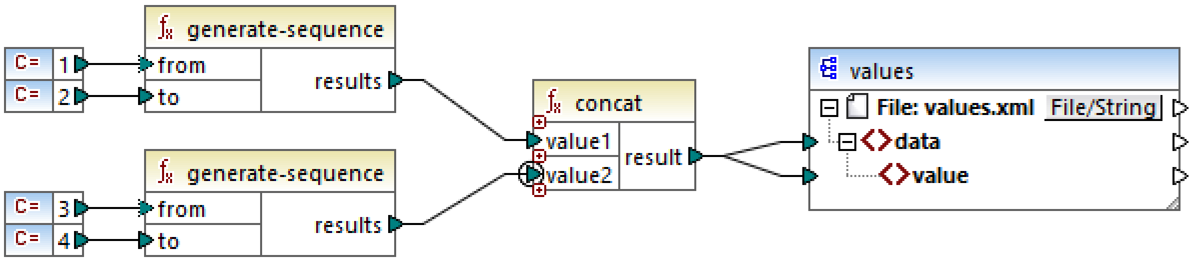
First, let's run the mapping without setting a priority context. The **concat** function starts evaluating the top sequence first, so it combines values in the following order:

- 1 with 3
- 1 with 4
- 2 with 3
- 2 with 4

This is reflected in the mapping output as well:

```
<data>
    <value>13</value>
    <value>14</value>
    <value>23</value>
    <value>24</value>
</data>
```

If you right-click the second input parameter and select **Priority Context** from the context menu, it will become the priority context. As illustrated below, the priority context input is encircled.



This time, the second input parameter will be evaluated first. The `concat` function will still concatenate the same values, but this time it will process the sequence `3,4` first. Consequently, the output becomes:

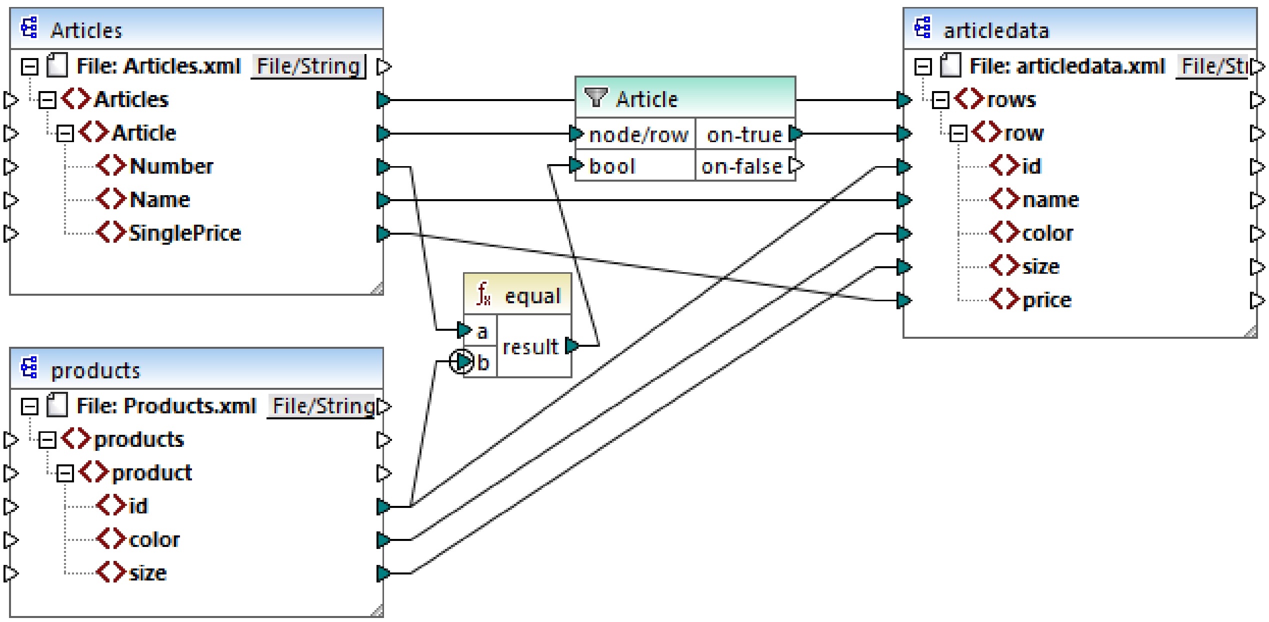
```
<data>
    <value>13</value>
    <value>23</value>
    <value>14</value>
    <value>24</value>
</data>
```

So far, you have seen only the theoretical part behind priority context. For a more practical scenario, see Example: Filter with priority context 410.

7.3.3.1 Example: Filter with priority context

When a function is connected to a filter, priority context affects not only the function itself, but also the evaluation of the filter. The mapping below illustrates a typical case when it's required to set a priority context in order to get the correct output. You can find this mapping at the following path: **<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\FilterWithPriority.mfd**.

Note: This mapping uses XML components, but the same logic as described below applies for all other component types in MapForce, including EDI, JSON, and so on.



The aim of the mapping above is to copy people data from **Articles.xml** into a new XML file with a different schema, **articledata.xml**. At the same time, the mapping should look up the details of each article in the **Products.xml** file and join them to the respective article record. Note that each record in **Articles.xml** has a **Number** and each record in **Products.xml** has an **id**. If these two values are equal, then all the other values (**Name**, **SinglePrice**, **color**, **size**) should be copied to the same **row** in the target.

This goal has been accomplished by adding a filter. Each filter requires a Boolean condition as input; only those nodes/rows that satisfy the condition will be copied over to the target. For this purpose, there is an **equal** function on the mapping. The **equal** function checks if the article number and product ID are equal in both sources. The result is then supplied as input to the filter. If **true**, then the **Article** item is copied to the target.

Notice that a priority context has been defined on the second input parameter of the second **equal** function. In this mapping, the priority context makes a big difference, and not setting it will result in incorrect mapping output.

Initial mapping: No priority context

Here is the mapping logic without priority context:

- According to the general mapping rule, for each **Article** that satisfies the filter condition, a new **row** is created in the target. The connection between **Article** and **row** (via the function and filter) takes care of this part.
- The filter checks the condition for each article. To do this, it iterates through all products, and brings multiple products in the current context.
- To populate the **id** on the target side, MapForce follows the general rule (for each item in the source, create an item in the target). However, as explained above, all products from **Products.xml** are in the current context. There is no connection between **product** to anywhere else in the target so as to read the **id** of a specific product only. As a consequence, multiple **id** elements will be created for each **Article** in the target. The same happens with **color** and **size**.

To summarize: items from **Products.xml** have the filter's context (which must iterate through each product); therefore, the **id**, **color**, and **size** values will be copied to each target **row** as many times as there are products in the source file, and generate incorrect output like the one below:

```
<rows>
    <row>
        <id>1</id>
        <id>2</id>
        <id>3</id>
        <name>T-Shirt</name>
        <color>red</color>
        <color>blue</color>
        <color>green</color>
        <size>10</size>
        <size>20</size>
        <size>30</size>
        <price>25</price>
    </row>
</rows>
```

Solution A: Use priority context

The problem above was solved by adding a priority context to the function that computes the filter's Boolean condition.

Specifically, if the second input parameter of the `equal` function is designated as priority context, the sequence incoming from **Products.xml** is prioritized. This translates to the following mapping logic:

- For each product, populate input **b** of the `equal` function (in other words, prioritize **b**). At this stage, the details of the current product are in context.
- For each article, populate input **a** of the `equal` function and check if the filter condition is true. If yes, then put the article details as well into the current context.
- Next, copy the article and product details from the current context to the respective items in the target.

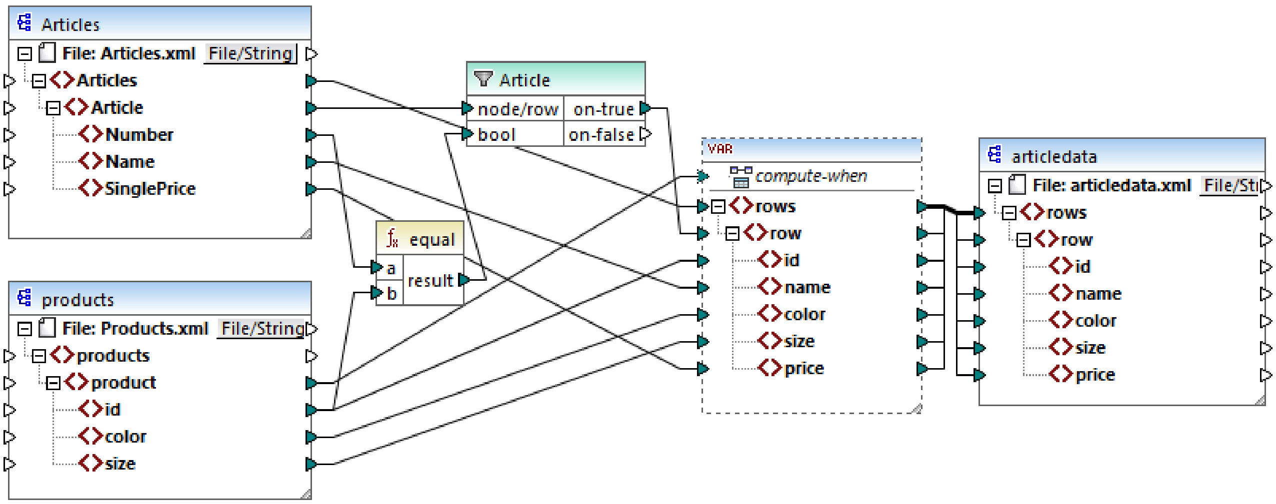
The mapping logic above produces correct output, for example:

```
<rows>
    <row>
        <id>1</id>
        <name>T-Shirt</name>
        <color>red</color>
        <size>10</size>
        <price>25</price>
    </row>
</rows>
```

Solution B: Use a variable

As an alternative solution, you could bring each article and product that matches the filter's condition into the same context with the help of an intermediate variable. Variables are suitable for scenarios like this one because they let you store data temporarily on the mapping, and thus help you change the context as necessary.

For scenarios like this one, you can add to the mapping a variable that has the same schema as the target component. On the **Insert** menu, click **Variable**, and supply the **articledata.xsd** schema as structure when prompted.



In the mapping above, the following happens:

- Priority context is not used any longer. There is a variable instead, which has the same structure as the target component.
- As usual, the mapping execution starts from the target root node. Before populating the target, the mapping collects data into the variable.
- The variable is computed in the context of each product. This happens because there is a connection from **product** to the **compute-when** input of the variable.
- The filter condition is thus checked in the context of each product. Only if the condition is true will the variable's structure be populated and passed on to the target.

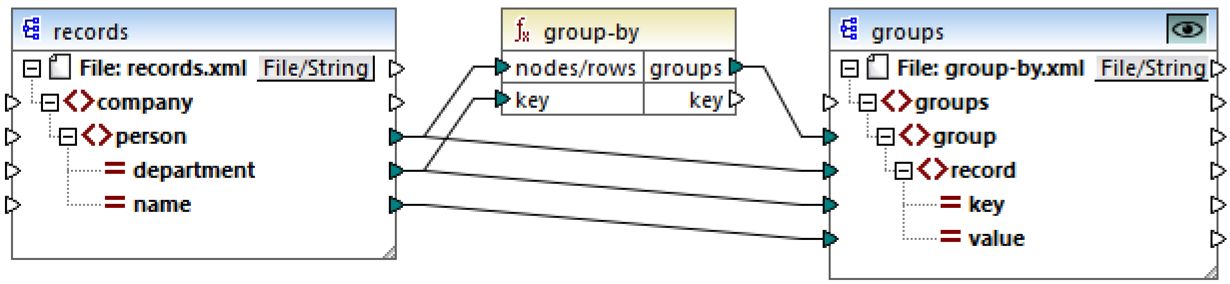
7.3.4 Multiple target components

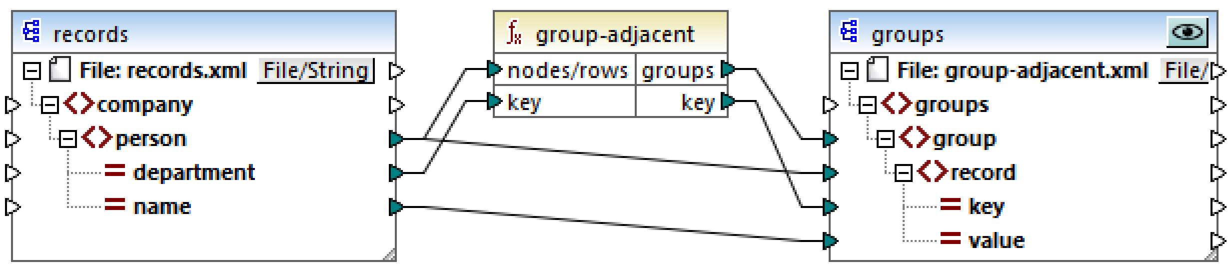
A mapping may have multiple source and target components. When there are multiple target components, you can preview only one component output at a time in MapForce, the one that you indicate by clicking the  **Preview** button. In other execution environments (MapForce Server or generated code), all of the target components will be executed sequentially, and the respective output of each component will be produced.

By default, target components are processed from top to bottom and from left to right. If necessary, you can influence this order by changing the position of target components in the mapping window. The point of reference is each component's top left corner. Note the following:

- If two components have the same vertical position, then the leftmost takes precedence.
- If two components 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.

For an example of how this works, open the following demo mapping:
<Documents>\Altova\MapForce2023\MapForceExamples\Tutorial\GroupingFunctions.mfd. This mapping consists of multiple source and multiple target components; only a fragment is shown below.





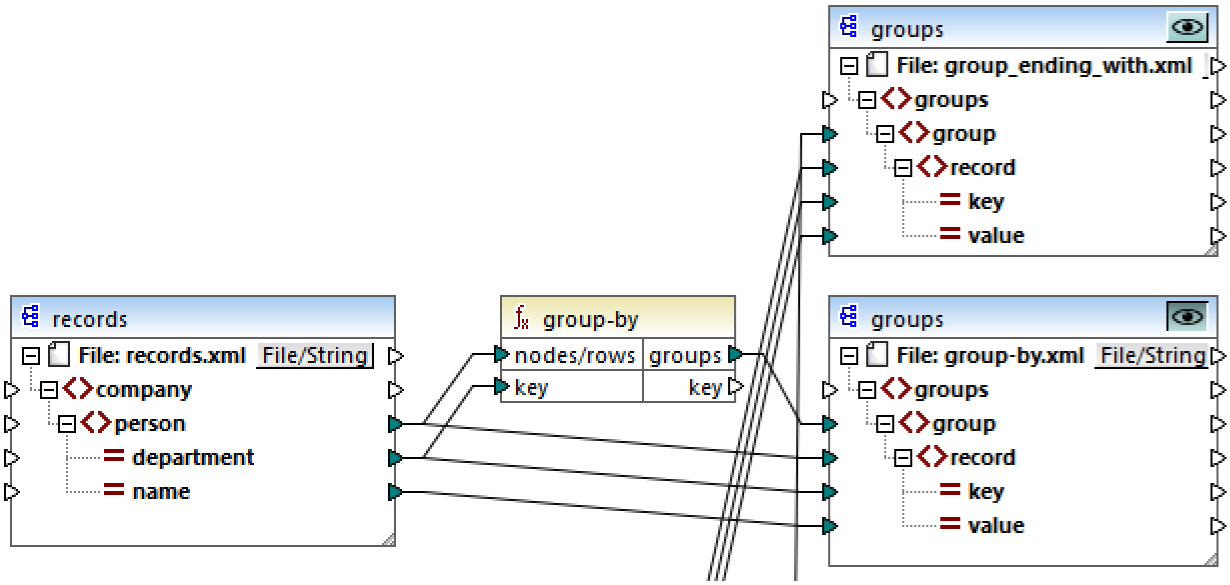
According to the rules, the default processing order of this mapping in MapForce Server and in generated code is from top to bottom. You can check that this is the case by generating XSLT 2.0 code, for example.

1. On the **File** menu, click **Generate code in | XSLT 2.0**.
2. When prompted, select a target directory for the generated code.

After generation, the target directory includes several XSLT files and a **DoTransform.bat** file. The latter can be executed by RaptorXML Server (requires a separate license). The **DoTransform.bat** file processes components in the same order as they were defined on the mapping, from top to bottom. This can be verified by looking at the `--output` parameter of each transformation.

```
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-by.xml" --xml-
validation-error-as-warning=true %* "MappingMapTogroups.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-adjacent.xml" --
xml-validation-error-as-warning=true %* "MappingMapTogroups2.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-into-blocks.xml" --
xml-validation-error-as-warning=true %* "MappingMapTogroups3.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records-v2.xml" --output="group-starting-
with.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups4.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records-v3.xml" --output="group_ending_with.xml"
--xml-validation-error-as-warning=true %* "MappingMapTogroups5.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
```


The last transformation produces an output file called **group-ending-with.xml**. Let's now move this target component on the mapping to the very top:



If you now generate the XSLT 2.0 code again, the processing order changes accordingly:

```
RaptorXML xslt --xslt-version=2 --input="records-v3.xml" --output="group_ending_with.xml"
--xml-validation-error-as-warning=true %* "MappingMapTogroups.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-by.xml" --xml-
validation-error-as-warning=true %* "MappingMapTogroups2.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-adjacent.xml" --
xml-validation-error-as-warning=true %* "MappingMapTogroups3.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records.xml" --output="group-into-blocks.xml" --
xml-validation-error-as-warning=true %* "MappingMapTogroups4.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
RaptorXML xslt --xslt-version=2 --input="records-v2.xml" --output="group-starting-
with.xml" --xml-validation-error-as-warning=true %* "MappingMapTogroups5.xslt"
IF ERRORLEVEL 1 EXIT/B %ERRORLEVEL%
```

In the code listing above, the first call now produces **group-ending-with.xml**.

You can change the processing order in a similar way in other code languages and in compiled MapForceServer execution files (.mfx).

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 components 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.

7.4 Processing Multiple Input or Output Files

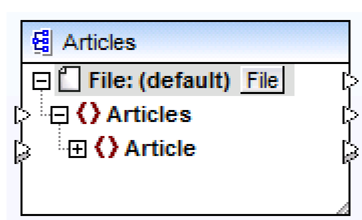
You can configure MapForce to process multiple files (for example, all files in a directory) when the mapping runs. Using this feature, you can solve tasks such as:

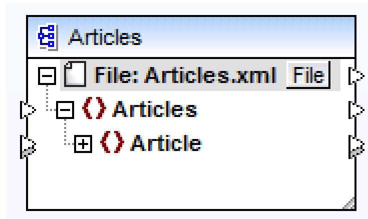
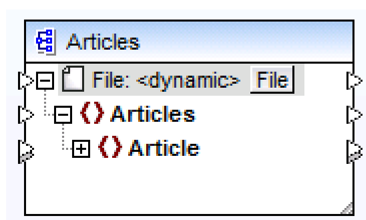
- Supply to the mapping a list of input files to be processed
- Generate as mapping output a list of files instead of a single output file
- Generate a mapping application where both the input and output file names are defined at runtime
- Convert a set of files to another format
- Split a large file 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 [71]). 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 a semi-colon (;), followed by the name of the output file.

	If the name of the input is the same with that of the output file, it is displayed as name of the root node.  Note that you can select either this option or the *Use Dynamic File Names Supplied by Mapping* option.
Use Dynamic File Names Supplied by Mapping	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. 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. When this option is selected, the name of the root node is displayed as **File: <dynamic>**.  This option is mutually exclusive with the *Use File Names from Component Settings* option.

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**
- JSON files**
- Protocol Buffers files**

** 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)	*
XSLT 3.0	*	*(1)	*
C++	*	*	*
C#	*	*	*
Java	*	*	*
BUILT-IN	*	*	*

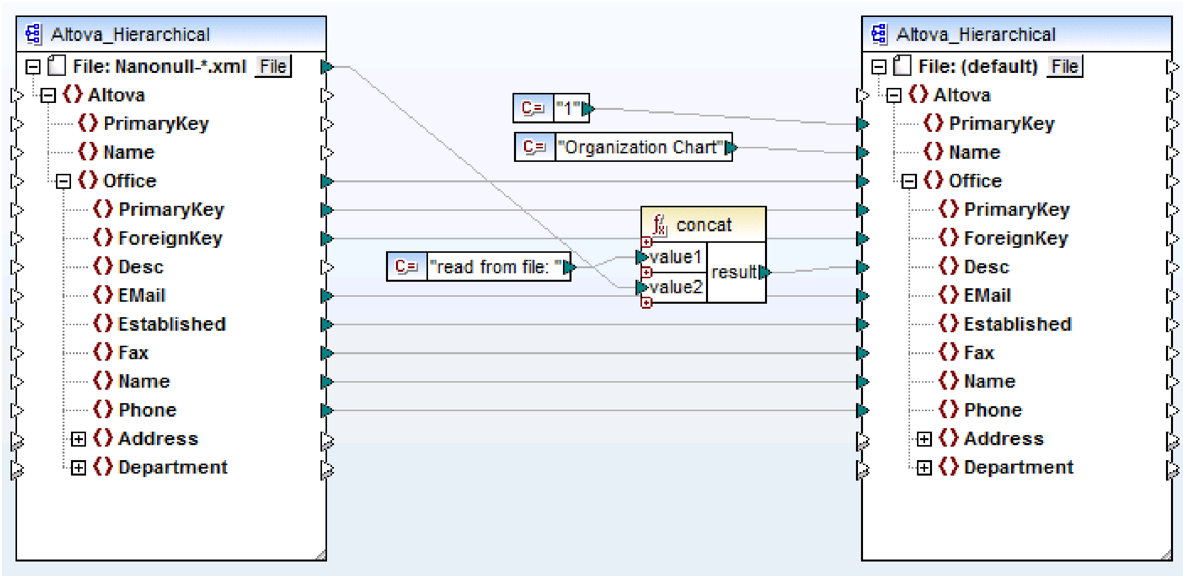
Legend:

*	Supported
(1)	XSLT 2.0, XSLT 3.0, and XQuery use the `fn:collection` function. The implementation in the Altova XSLT 2.0, XSLT 3.0, and XQuery engines resolves wildcards. Other engines may behave differently.

7.4.1 Mapping Multiple Input Files to a Single Output File

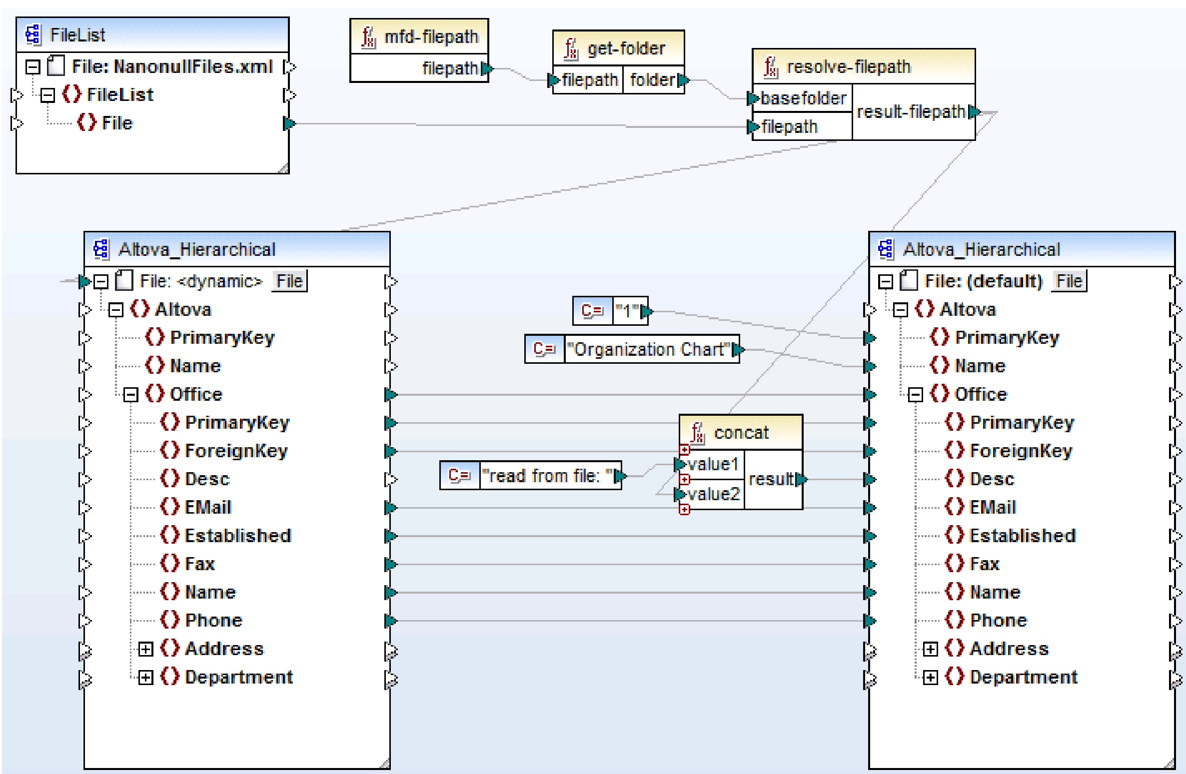
To process multiple input files, do one of the following:

- 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.

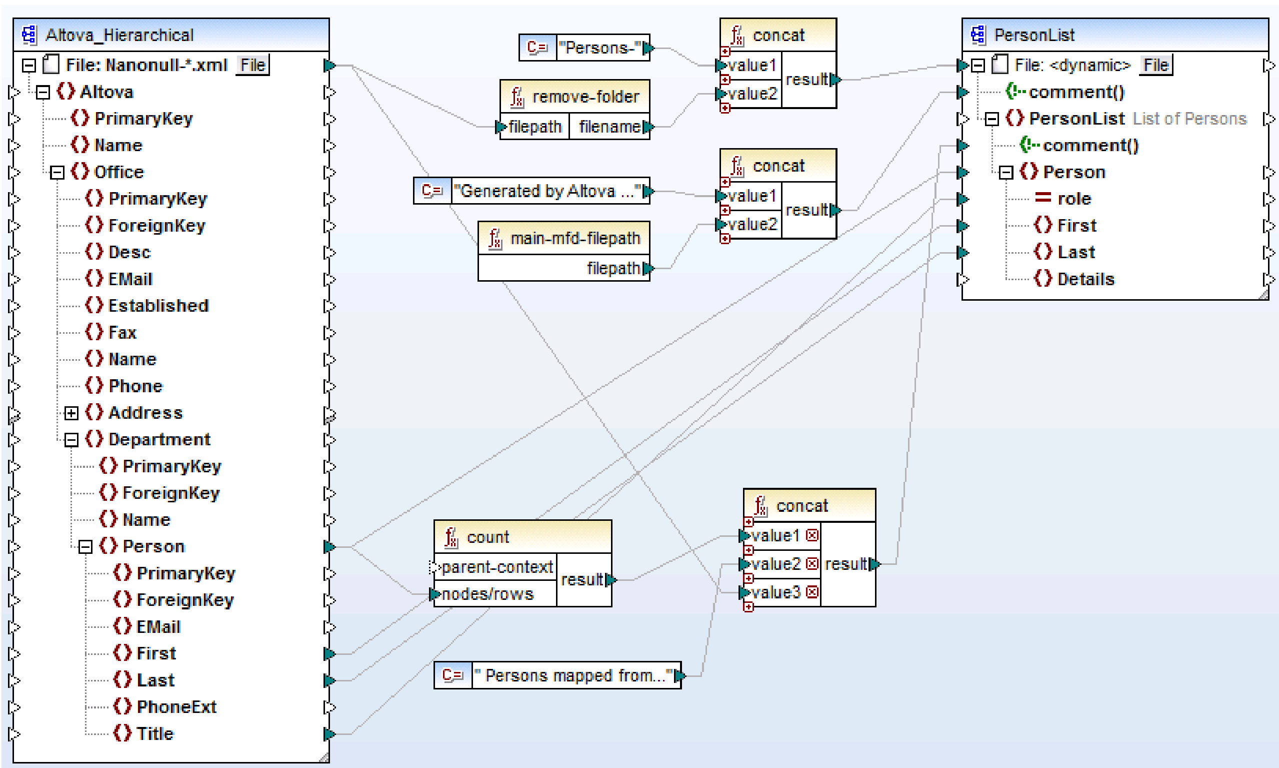


MergeMultipleFiles_List.mfd (MapForce Basic Edition)

7.4.2 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 103).

7.4.3 Supplying File Names as Mapping Parameters

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

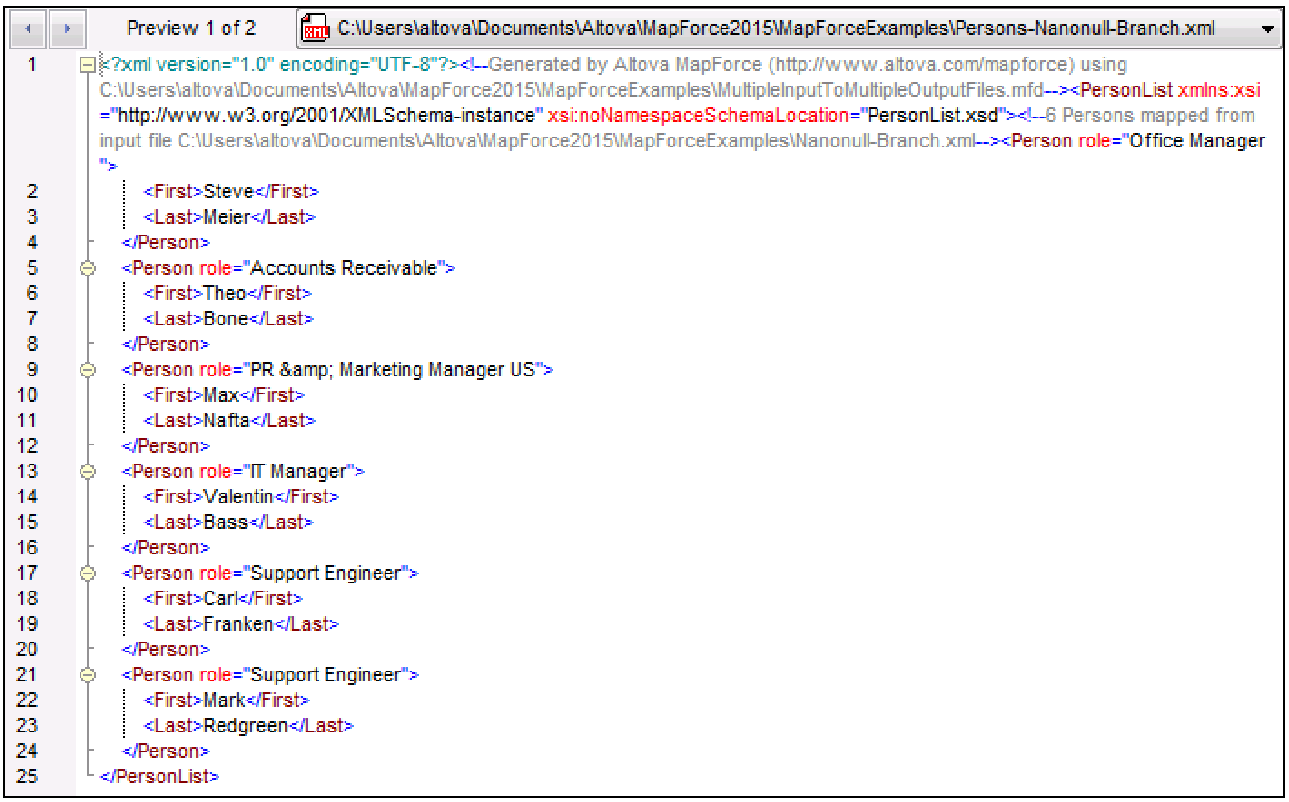
1. Add a simple input component to the mapping (On the **Function** menu, click **Insert Input**). For more information about such components, see Supplying Parameters to the Mapping 139.

2. Click the **File** (File) or **File/String** (File/String) button of the source component and select **Use Dynamic File Names Supplied by Mapping**.
3. Connect the simple 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](143).

7.4.4 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

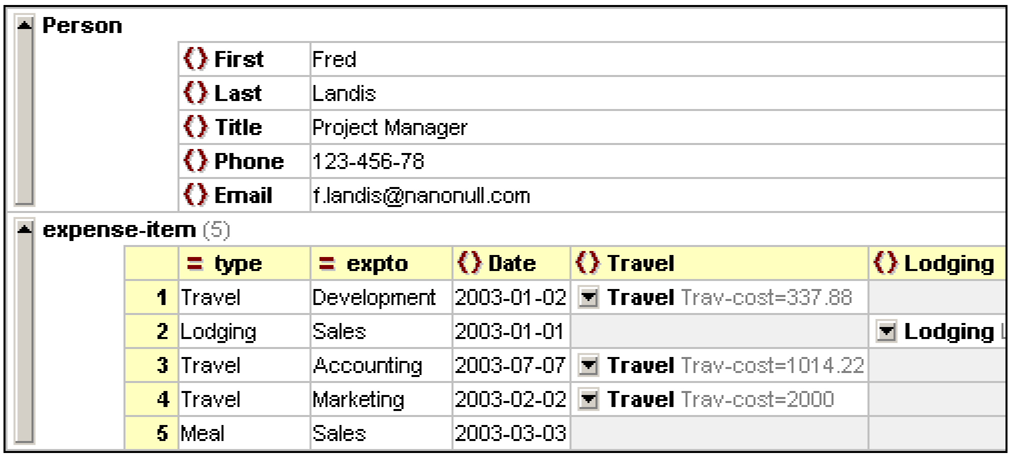
To save the generated output files, do one of the following:

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

7.4.5 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\MapForce2023\MapForceExamples\Tutorial\Tut-ExpReport-dyn.mfd**.

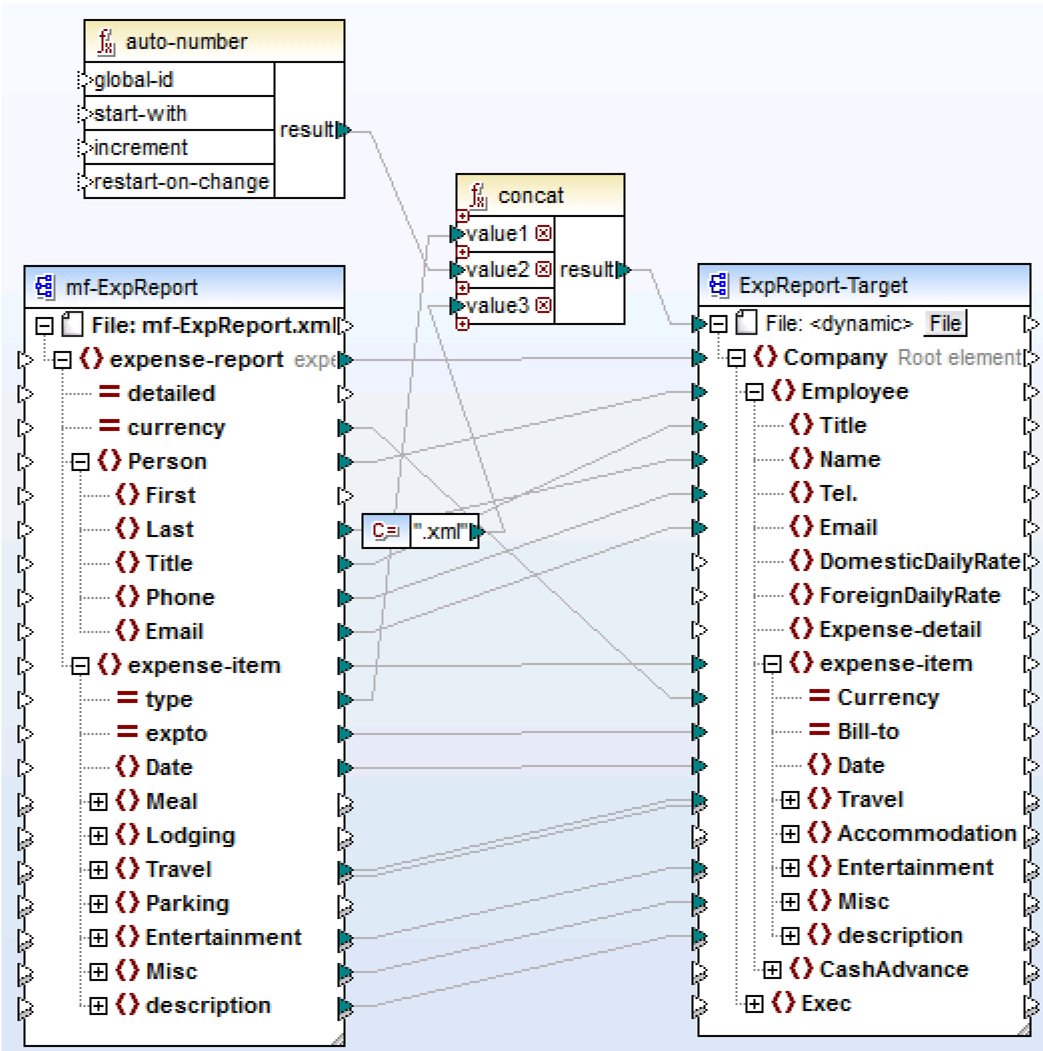
The source XML file (available in the same folder as the mapping) consists of the expense report for a person called "Fred Landis" and contains five expense items of different types. The aim of the example is to generate a separate XML file for each of the expense items listed below.



mf-ExpReport.xml (as shown in XMLSpy Grid view)

As the `type` attribute defines the specific expense item type, this is the item we will use to split up the source file. To achieve the goal of this example, do the following:

1. Insert a **`concat`** function (you can drag it from the **core | string functions** library of the Libraries pane).
2. Insert a constant (on the **Insert** menu, click **Constant**) and enter ".xml" as its value.
3. Insert the **`auto-number`** function (you can drag it from the **core | generator functions** library of the Libraries pane).
4. Click the **File** (File) or **File/String** (File/String) button of the target component and select **Use Dynamic File Names Supplied by Mapping**.
5. Create the connections as shown below and then click the **Output** tab to see the result of the mapping.



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.

8 Automation with Altova Products

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, XSLT 3.0, or XQuery. See Automation with RaptorXML Server 426.
- *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 427.

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 is 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.
- 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 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 96 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.
`/XSLT3`	Generates XSLT 3.0 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>`	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 Altova Global Resources 429. The option `/GLOBALRESOURCEFILE` specifies the path to a Global Resource .xml file. Note that, if `/GLOBALRESOURCEFILE` is set, then `/GLOBALRESOURCECONFIG` must also be set.
`/GLOBALRESOURCECONFIG <config>`	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.
`/LOG <logfilename>`	Generates a log file at the specified path. `<logfilename>` 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 specify only the file name, then the file will be placed in the current directory of the Windows command prompt.

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>
```


9 Altova Global Resources

Altova Global Resources are aliases for file, folder, and database resources. Each alias can have multiple configurations, and each configuration maps to a single resource. Therefore, when you use a global resource, you can switch between its configurations. For example, you could create a database resource with two configurations: `development` and `production`. Depending on your goals, you can switch between these configurations.

Global resources can be used across different Altova applications (*see subsection below*).

Global resources in other Altova products

When stored as global resources, files, folders, and database connection details become reusable across multiple Altova applications. For example, if you often need to open the same file in multiple Altova desktop applications, you can define this file as a global resource. If you need to change the file path, you will need to change it only in one place. Currently, global resources can be defined and used in the following Altova products:

- Altova Authentic
- DatabaseSpy
- MobileTogether Designer
- MapForce
- StyleVision
- XMLSpy
- FlowForce Server
- MapForce Server
- RaptorXML Server/RaptorXML+XBRL Server

In this section

This section explains how to create and configure different types of global resources. The section is organized into the following topics:

- Global Resource Setup Part 1 430
- Global Resource Setup Part 2 432
- XML Files as Global Resources 435
- Folders as Global Resources 437

9.1 Global Resource Setup Part 1

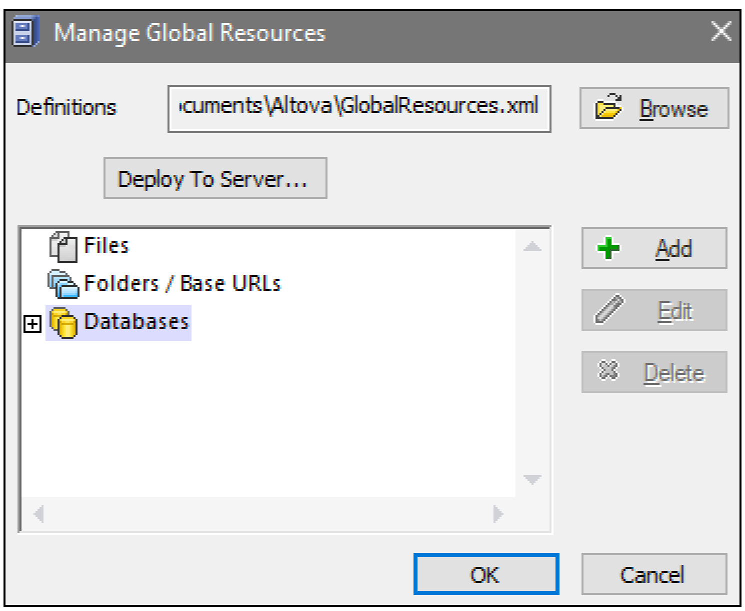
The global resource setup consists of two parts: (i) creating a global resource in the **Manage Global Resources** dialog box (*see below*) and (ii) defining the properties of this global resource in the **Global Resource** dialog. The second part is discussed in the next topic 432.

Altova Global Resources are defined in the **Manage Global Resources** dialog, which can be accessed in two ways:

- Click the menu command **Tools | Global Resources**.
- Click the **Manage Global Resources** icon in the Global Resources toolbar (*screenshot below*).

Global Resources Definitions file

Information about global resources is stored in an XML file called the Global Resources Definitions file. This file is created when the first global resource is defined in the **Manage Global Resources** dialog box (*screenshot below*) and saved.



When you open the **Manage Global Resources** dialog box for the first time, the default location and name of the Global Resources Definitions file is specified in the *Definitions* text box (*see screenshot above*):

```
C:\Users\<username>\Documents\Altova\GlobalResources.xml
```

This file is set as the default Global Resources Definitions file for all Altova applications. A global resource can be saved from any Altova application to this file and will immediately be available to all other Altova applications as a global resource. To define and save a global resource to the Global Resources Definitions file, add the global resource in the **Manage Global Resources** dialog and click **OK** to save.

To select an already existing Global Resources Definitions file to be the active definitions file of a particular Altova application, browse for it via the **Browse** button of the *Definitions* text box (*see screenshot above*).

The **Manage Global Resources** dialog box also allows you to edit and delete existing global resources.

Notes:

- You can give any name to the Global Resources Definitions file and save it to any location accessible to your Altova applications. All you need to do in each application, is specify this file as the Global Resources Definitions file for that application (in the *Definitions* text box). The resources become global across Altova products when you use a single definitions file across all Altova products.

- You can also create multiple Global Resources Definitions files. However, only one of these can be active at any time in a given Altova application, and only the definitions contained in this file will be available to the application. The availability of resources can therefore be restricted or made to overlap across products as required.

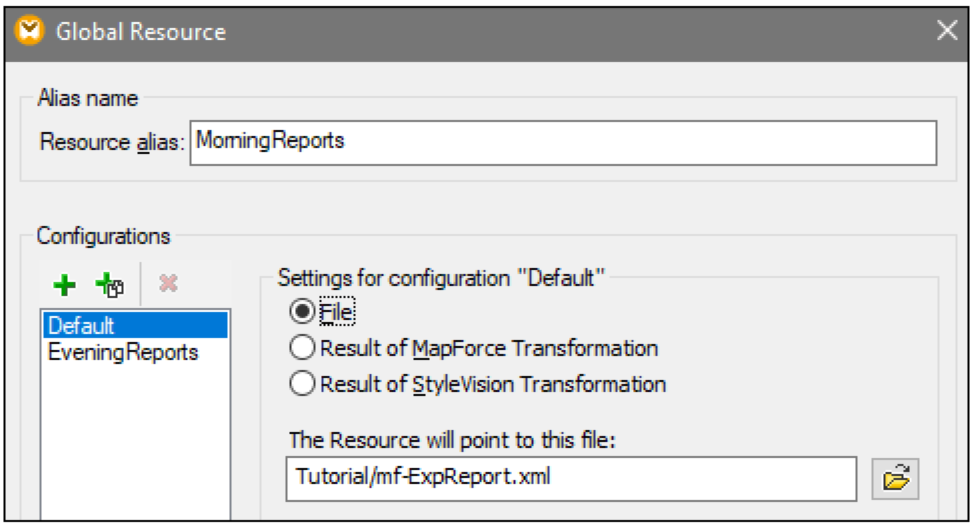
9.2 Global Resource Setup Part 2

The second part of the global resource setup consists in defining properties of a global resource in the **Global Resource** dialog box. The properties depend on the type of a global resource (*see subsections below*). You can access the **Global Resource** dialog box by clicking the **Add** button in the Manage Global Resources dialog box [430].

To find out more about setting up different types of global resources, see the following examples: XML Files as Global Resources [435], Folders as Global Resources [437].

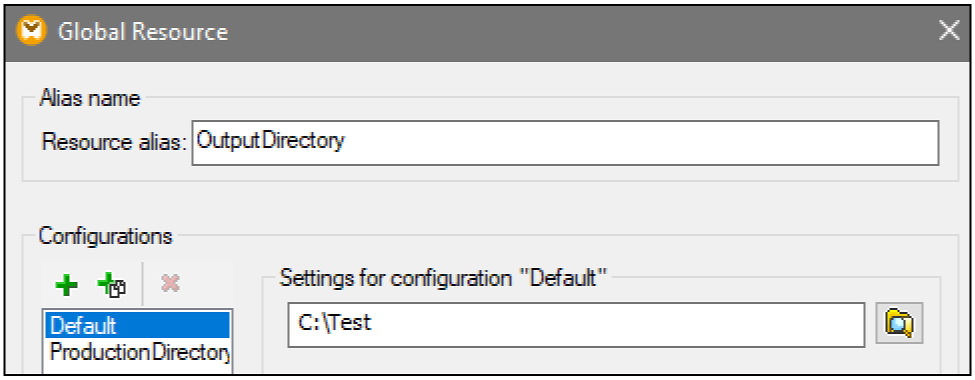
Files

The file-specific properties are shown in the **Global Resource** dialog box below. The setup has three major parts: (i) the name of the file, (ii) the location of this file, and (iii) the list of configurations defined for this file alias.



Folders

The folder-specific properties are shown in the **Global Resource** dialog box below. The setup has three major parts: (i) the name of the folder, (ii) the location of this folder, and (iii) the list of configurations defined for this folder alias.



Global Resource dialog icons

	*Add Configuration:* Pops up the Add Configuration dialog in which you enter the name of the configuration to be added.
	*Add Configuration as Copy:* Pops up the Add Configuration dialog in which you can enter the name of the configuration to be created as a copy of the selected configuration.
	*Delete:* Deletes the selected configuration.
	*Open:* Browse for the file to be created as the global resource.
	*Open:* Browse for the folder to be created as the global resource.

Global resource setup: General procedures

The broad procedure of creating and configuring global resources is described below:

1. Click the  toolbar button (**Manage Global Resource**). Alternatively, go to the **Tools** menu and click **Global Resources**.
2. Click **Add** and select the resource type you wish to create (file, folder, database). The **Global Resource** dialog box will appear.
3. Enter a descriptive name in the **Resource alias** text box (e.g., `InputFile`).
4. Setting up the default configuration depends on the type of the global resource: (i) For a file or folder, browse for the file or folder to which this resource should point by default; (ii) for 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 you run the mapping.

5. If you need an additional configuration (e.g., an additional output folder), click the button in the **Global Resource** dialog box, enter the name of this configuration, and specify the path to this configuration.
6. Repeat the previous step for each additional configuration required.

Note: Database connections are supported as global resources only in MapForce Professional and Enterprise editions.

9.3 XML Files as Global Resources

This topic explains how to use XML files as global resources. There are situations in which you may need to change an input XML file multiple times per day. For example, every morning you need to run a particular mapping and generate a report by using one XML file as a mapping input, and every evening the same report must be generated from another XML file. 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 (the so-called *file alias*). In this example, our file alias will have two configurations:

1. The `Default` configuration will supply a morning XML file as a mapping input.
2. The `EveningReports` configuration will supply an evening XML file as a mapping input.

To create and configure the file alias, take the steps below.

Step 1: Create a global resource

First, we need to create a file alias. Follow the instructions below:

1. Click the toolbar button (**Manage Global Resource**). Alternatively, go to the **Tools** menu and click **Global Resources**.
2. Click **Add | File** and enter a name in the **Resource alias** text box. In this example, we call our default configuration `MorningReports`.
3. Click the **Browse** button near the text field **The Resource will point to this file** and select **`Tutorial\mf-ExpReport.xml`**.
4. Click in the **Configurations** section and name the second configuration `EveningReports`.
5. Click **Browse** and select **`Tutorial\mf-ExpReport2.xml`**.

Step 2: Use the global resource in the mapping

Now we can use the newly created global resource in our mapping. To make the mapping read data from the global resource, take the steps below:

1. Open the **`Tutorial\Tut-ExpReport.mfd`** mapping.
2. Double-click the header of the source component to open the **Component Settings** dialog box.
3. Next to **Input XML file**, click **Browse**, then click **Global Resources** and select the file alias `MorningReports`. Click **Open**.
4. Open the **Component Settings** dialog box again: The input XML file path has now become **`altova://file_resource/MorningReports`**, which indicates that the path uses a global resource.

Step 3: Run the mapping with the desired configuration

You can now switch between the input XML files before running the mapping:

- To use **`mf-ExpReport.xml`** as an input, select the menu item **Tools | Active Configuration | Default**.
- To use **`mf-ExpReport2.xml`** as an input, select the menu item **Tools | Active Configuration | EveningReports**.

Alternatively, select the required configuration from the **Global Resources** drop-down list in the toolbar (*see screenshot below*).

To preview the mapping result with either configuration, click the **Output** pane.

9.4 Folders as Global Resources

This topic explains how to use folders as global resources. There are situations in which you may need to generate the same output in different directories. To make this possible, we will create a folder alias with two configurations:

1. The `Default` configuration will generate the output in **`C:\Test`**.
2. The `Production` configuration will generate the output in **`C:\Production`**.

To create and configure the folder alias, take the steps below.

Step 1: Create a global resource

First, we need to create a folder alias. Follow the instructions below:

1. Click the toolbar button (**Manage Global Resource**). Alternatively, go to the **Tools** menu and click **Global Resources**.
2. Click **Add | Folder** and enter a name in the **Resource alias** text box. In this example, we call our default configuration `OutputDirectory`.
3. Click the **Browse** button near the text field **Settings for configuration "Default"** and select **`C:\Test`**. Make sure this folder already exists on your operating system.
4. Click and enter a name of the second configuration. In this example, we call our second configuration `ProductionDirectory`.
5. Click **Browse** and select the **`C:\Production`** folder. Make sure that this folder already exists on your operating system.

Step 2: Use the global resource in the mapping

The next step is to make the mapping use the folder alias we have just created. Take the steps below:

1. Open the **`Tutorial\Tut-ExpReport.mfd`** mapping.
2. Double-click the header of the target component to open the **Component Settings** dialog box.
3. Click **Global Resources** and then click **Save**.
4. Save the output XML file as **`Output.xml`**. 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 switch between the output folders before running the mapping:

- To use **`C:\Test`** as the output directory, select the menu item **Tools | Active Configuration | Default**.
- To use **`C:\Production`** as the output directory, select the menu item **Tools | Active Configuration | ProductionDirectory**.

By default, the mapping output is written as a temporary file unless you explicitly configure MapForce to write output to permanent files. To configure MapForce so that it generates permanent files, do the following:

1. Go to the **Tools** menu and click **Options**.
2. In the **General** section, select the option **Write directly to final output files**.

10 Catalogs in MapForce

MapForce supports a subset of the OASIS XML catalogs mechanism. The catalog mechanism enables MapForce to retrieve commonly used schemas (as well as 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 in this section:

- How Catalogs Work 440
- Catalog Structure in MapForce 442
- Customizing Your Catalogs 444
- Environment Variables 446

For more information on catalogs, see the XML Catalogs specification.

10.1 How Catalogs Work

Catalogs can be used to redirect both DTDs and XML Schemas. While the concept behind the mechanisms of both cases is the same, the details are different and are explained below.

DTDs

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, its 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 map the **PUBLIC** identifier 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, then 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>
```

The catalog is searched for the **PUBLIC** identifier of this SVG file. 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. As a result, the lookup for the SVG DTD is redirected to the URL **schemas/svg/svg11.dtd** (which is relative to the catalog file). This is a local file that will be used as the DTD for the SVG file. If there is no mapping for the **Public** ID in the catalog, then the URL in the XML document will be used (in the SVG fie example above, this is the Internet URL: **http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd**).

XML Schemas

In MapForce, you can also use catalogs with **XML Schemas**. In the XML instance file, the reference to the schema will occur in the **xsi:schemaLocation** attribute of the XML document's top-level element. For example,

```
xsi:schemaLocation="http://www.xmlspy.com/schemas/orgchart OrgChart.xsd"
```

The value of the **xsi:schemaLocation** attribute has two parts: a namespace part (green above) and a URI part (highlighted). The namespace part is used in the catalog to map to the alternative resource. For example, the following catalog entry redirects the schema reference above to a schema at an alternative location.

```
<uri name="http://www.xmlspy.com/schemas/orgchart" uri="C:\MySchemas\OrgChart.xsd"/>
```

Normally, the URI part of the **xsi:schemaLocation** attribute's value is a path to the actual schema location. However, if the schema is referenced via a catalog, the URI part need not point to an actual XML Schema but must exist so that the lexical validity of the **xsi:schemaLocation** attribute is maintained. A value of `foo`, for example, would be sufficient for the URI part of the attribute's value to be valid.

10.2 Catalog Structure in MapForce

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 of which is referenced in a **`nextCatalog`** element. These catalog files are looked up and the URIs in them are resolved according to their mappings.

Listing of RootCatalog.xml

```
<?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"/>
  <!-- Include all catalogs under common schemas folder on the first directory level -->
  <nextCatalog spy:recurseFrom="%CommonSchemasFolder%" catalog="catalog.xml"
spy:depth="1"/>
  <nextCatalog spy:recurseFrom="%ApplicationWritableDataFolder%/pkgs/.cache"
catalog="remapping.xml" spy:depth="0"/>
  <nextCatalog catalog="CoreCatalog.xml"/>
</catalog>
```

The listing above references a custom catalog (named **`CustomCatalog.xml`**) and a set of catalogs that locate commonly used schemas (such as W3C XML Schemas and the SVG schema).

- **`CustomCatalog.xml`** is located in your Personal Folder (located via the variable **`%PersonalFolder%`**). It is a skeleton file in which you can create your own mappings. You can add mappings to **`CustomCatalog.xml`** for any schema you require that is not addressed by the catalog files in the Common Schemas Folder. Do this by using the supported elements of the OASIS catalog mechanism (*see next section*).
- The Common Schemas Folder (located via the variable **`%CommonSchemasFolder%`**) contains a set of commonly used schemas. Inside each of these schema folders is a **`catalog.xml`** file that maps public and/or system identifiers to URIs that point to locally saved copies of the respective schemas.
- **`CoreCatalog.xml`** is located in the MapForce application folder, and is used to locate schemas and stylesheets used by MapForce-specific processes, such as StyleVision Power Stylesheets which are stylesheets used to generate Altova's Authentic View of XML documents.

Location variables

The variables that are used in **`RootCatalog.xml`** (*listing above*) have the following values:

`%PersonalFolder%`	Personal folder of the current user, for example `C:\Users\<name>\Documents`
`%CommonSchemasFolder%`	`C:\ProgramData\Altova\Common2023\Schemas`
`%ApplicationWritableDataFolder%`	`C:\ProgramData\Altova`

Location of catalog files and schemas

Note the locations of the various catalog files.

- **`RootCatalog.xml`** and **`CoreCatalog.xml`** are in the MapForce application folder.
- **`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 Common Schemas Folder.

10.3 Customizing Your Catalogs

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"/>`

Note the following points:

- 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.
- A URI can be mapped to another URI using the `uri` element.
- The `rewriteURI` and `rewriteSystem` 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.

From release 2014 onwards, MapForce adheres closely to the XML Catalogs specification (OASIS Standard V1.1, 7 October 2005) specification. This specification strictly separates external-identifier look-ups (those with a Public ID or System ID) from URI look-ups (URIs that are not Public IDs or System IDs). Namespace URIs must therefore be considered simply URIs—not Public IDs or System IDs—and must be used as URI look-ups rather than external-identifier look-ups. In MapForce versions prior to version 2014, schema namespace URIs were translated through **`<public>`** mappings. From version 2014 onwards, **`<uri>`** mappings have to be used.

Prior to v2014: `<public publicID="http://www.MyMapping.com/ref" uri="file:///C:/MyDocs/Catalog/test.xsd"/>`
V-2014 onwards: `<uri name="http://www.MyMapping.com/ref" uri="file:///C:/MyDocs/Catalog/test.xsd"/>`

How MapForce finds a referenced schema

A schema is referenced in an XML document via the **`xsi:scemaLocation`** attribute (*shown below*). The value of the **`xsi:schemaLocation`** attribute has two parts: a namespace part (green) and a URI part (highlighted).

```
xsi:schemaLocation="http://www.xmlspy.com/schemas/orgchart OrgChart.xsd"
```

Given below are the steps, followed sequentially by MapForce, to find a referenced schema. The schema is loaded at the first successful step.

1. Look up the catalog for the URI part of the **`xsi:schemaLocation`** value. If a mapping is found, including in **`rewriteURI`** mappings, use the resulting URI for schema loading.
2. Look up the catalog for the namespace part of the **`xsi:schemaLocation`** value. If a mapping is found,

including in **rewriteURI** mappings, use the resulting URI for schema loading.
3. Use the URI part of the **xsi:schemaLocation** value for schema loading.

XML Schema specifications

XML Schema specification information is built into MapForce and the validity of XML Schema (.xsd) documents is checked against this internal information. In an XML Schema document, therefore, no references should be made to any schema that defines the XML Schema specification.

The **catalog.xml** file in the %AltovaCommonSchemasFolder%\Schemas\schema folder contains references to DTDs that implement older XML Schema specifications. You should not validate your XML Schema documents against 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.

10.4 Environment 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:

`%PersonalFolder%`	Full path to the Personal folder of the current user, for example `C:\Users\<name>\Documents`
`%CommonSchemasFolder%`	`C:\ProgramData\Altova\Common2023\Schemas`
`%ApplicationWritableDataFolder%`	`C:\ProgramData\Altova`
`%AltovaCommonFolder%`	C:\Program Files\Altova\Common2023
`%DesktopFolder%`	Full path to the Desktop folder of the current user.
`%ProgramMenuFolder%`	Full path to the Program Menu folder of the current user.
`%StartMenuFolder%`	Full path to Start Menu folder of the current user.
`%StartUpFolder%`	Full path to Start Up folder of the current user.
`%TemplateFolder%`	Full path to the Template folder of the current user.
`%AdminToolsFolder%`	Full path to the file system directory that stores administrative tools of the current user.
`%AppDataFolder%`	Full path to the Application Data folder of the current user.
`%CommonAppDataFolder%`	Full path to the file directory containing application data of all users.
`%FavoritesFolder%`	Full path of the Favorites folder of the current user.
`%PersonalFolder%`	Full path to the Personal folder of the current user.
`%SendToFolder%`	Full path to the SendTo folder of the current user.
`%FontsFolder%`	Full path to the System Fonts folder.
`%ProgramFilesFolder%`	Full path to the Program Files folder of the current user.
`%CommonFilesFolder%`	Full path to the Common Files folder of the current user.
`%WindowsFolder%`	Full path to the Windows folder of the current user.
`%SystemFolder%`	Full path to the System folder of the current user.
`%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.

11 Menu Commands

This reference section contains a description of the MapForce menu commands. The following menu commands are available:

- File 448
- Edit 451
- Insert 452
- Component 455
- Connection 457
- Function 458
- Output 459
- View 461
- Tools 463
- Window 473
- Help 474

11.1 File

This topic lists all the menu commands available in the **File** menu.

⊟ New

Creates a new mapping document. In Professional and Enterprise editions, you can also create a mapping project (`.mfp`).

⊟ Open

Opens the previously saved mapping design (`.mfd`). In Professional and Enterprise editions, you can also open a mapping project (`.mfp`).

⊟ Save/Save As/Save All

The **Save** option saves the currently active mapping with its current name. The **Save As** option allows you to save the currently open mapping with a different name. The **Save All** command saves all currently open mapping files.

⊟ Reload

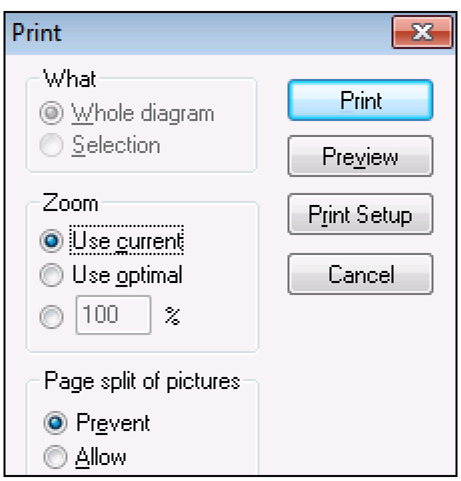
Reloading the currently active mapping reverts your last changes.

⊟ Close/Close All

The **Close** command closes the currently active mapping. The **Close All** command closes all currently open mappings. You are asked if you want to save any of the unsaved files.

⊟ Print/Print Preview/Print Setup

The **Print** command opens the **Print** dialog box (*see below*) that enables you to print out your mapping. **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 graphics to be split over several pages or not.



The **Preview** command opens the same **Print** dialog box with the same settings as described above. The **Print Setup** command opens the **Print Setup** dialog box in which you can select a printer and configure paper settings.

⊟ Validate Mapping

The **Validate Mapping** command checks whether all mappings are valid and displays relevant information messages, warnings, and errors. For details, see Validation 94.

⊟ Mapping Settings

Opens the Mapping Settings dialog box 103 where you can define document-specific settings.

⊟ Open Credentials Manager (*Enterprise Edition*)

Opens **Credentials Manager** that allows you to manage credentials required in mappings that perform basic HTTP authentication or OAuth 2.0 authorization.

⊟ Generate Code in Selected Language/Generate Code in

Generates code in the currently selected language. For more information about the available transformation languages, see Transformation Languages 16.

Generate Code in | XSLT (XSLT2, XSLT3)
This command generates the XSLT file(s) from the source file(s). Selecting this option opens the **Browse for Folder** dialog box in which you need to select the location of the XSLT file. The name of the generated XSLT file(s) is defined in the Mapping Settings dialog box 103.

Generate Code in | XQuery | Java | C# | C++ (Professional and Enterprise editions)
This command generates source code in one of the selected transformation languages. Selecting this option opens the **Browse for Folder** dialog box in which you need to select the location of the generated files. The names of the generated application files (as well as the **`*.csproj`** C# project file, **`*.sln`** solution file, and **`*.vcproj`** visual C++ project file) are defined in the Mapping Settings 103 dialog box.

The file name created by the executed code appears in the **Output XML File** box of the Component Settings 107 dialog box if the target is an XML/Schema document.

⊟ Compile to MapForce Server Execution File (*Professional and Enterprise editions*)

Generates a file that can be executed by MapForce Server to run the mapping transformation.

⊟ Deploy to FlowForce Server (*Professional and Enterprise editions*)

Deploys the currently active mapping to FlowForce Server.

⊟ Generate Documentation (*Professional and Enterprise editions*)

Generates documentation of your mapping projects in great detail in various output formats.

⊟ Recent files

Displays the list of the most recently opened files.

⊟ Exit

Exits the application. You are asked if you want to save any unsaved files.

11.2 Edit

This topic lists all the menu commands available in the **Edit** menu. Most of the commands in this menu become active when you view the result of a mapping in the **Output** pane or preview code, for example, in the **XSLT** pane.

Undo

MapForce has an unlimited number of undo steps that you can use to retrace your mapping steps. You can also use the toolbar command to undo actions.

Redo

The redo command allows you to redo previously undone actions. You can step backward and forward through the undo history using both these commands. You can also use the toolbar command to redo actions.

Find

Allows you to search for specific text in any of the **XQuery** (*Professional and Enterprise editions*), **XSLT**, **XSLT2**, **XSLT3**, and **Output** panes. You can also do a search using the toolbar command.

Find Next

Searches for the next occurrence of the same search string. You can also search for the next occurrence using the toolbar button.

Find Previous

Searches for the previous occurrence of the same search string. Searching for the previous occurrence is also possible with the toolbar command.

Cut/Copy/Paste/Delete

The standard windows Edit commands that allow you to cut, copy, paste, and delete any components or functions visible in the mapping window.

Select all

Selects all components in the **Mapping** pane or the text/code in the **XQuery** (*Professional and Enterprise editions*), **XSLT**, **XSLT2**, **XSLT3**, and **Output** panes.

11.3 Insert

This topic lists all the menu commands available in the **Insert** menu.

XML Schema/File

Adds an XML schema or instance file to the mapping. If you select an XML file without a schema reference, MapForce can generate a matching XML schema[106]. If you select an XML schema file, you are prompted to include an XML instance file which supplies the data for preview. You can also add an XML/XSD file via the toolbar command.

Database (*Professional and Enterprise editions*)

Adds a database component. You can also add a database component via the toolbar command. In MapForce Enterprise Edition, you can also add NoSQL databases as components.

EDI (*Enterprise Edition*)

Adds an EDI document. You can also add an EDI component via the toolbar command.

Text File (*Professional and Enterprise editions*)

Adds a flat file document such as CSV or a fixed-length text file. You can also add a text file via the toolbar command. MapForce Enterprise Edition also allows you to process text files with FlexText.

Web Service Function (*Enterprise Edition*)

Adds a call to a Web service. You can also add a Web service via the toolbar button.

Excel 2007+ File (*Enterprise Edition*)

Adds a Microsoft Excel 2007+ (`.xlsx`) file. If you do not have Excel 2007+ installed on your machine, you can still map to or from Excel 2007+ files. In this case, you cannot preview the result in the **Output** pane, but you can still save the result. You can also add an Excel file via the toolbar command.

XBRL Document (*Enterprise Edition*)

Adds an XBRL instance or taxonomy document. You can also add an XBRL component via the toolbar command.

JSON Schema/File (*Enterprise Edition*)

Adds a JSON schema or file. You can also add a JSON component via the toolbar command.

Protocol Buffers File (*Enterprise Edition*)

Adds a binary file encoded in Protocol Buffers format. You can also add a file in Protocol Buffers format via the toolbar command.

Insert Input

Simple-Input components can be used as input parameters that are relevant to the entire mapping or only in the context of user-defined functions. For more information, see Simple Input[139] and Parameters in UDFs[204]. You can also insert a Simple-Input component using the toolbar command.

Insert Output

Simple-Output components can be used as output components in mappings and as output parameters of user-defined functions. For more information, see Simple Output[146] and Parameters in UDFs[204]. You can also insert a Simple-Output component using the toolbar command.

Constant

Inserts a constant which supplies fixed data to an input connector. You can select the following types of data: `String`, `Number` and `All other`. You can also insert a constant using the toolbar command.

Variable

Inserts a variable[150], which is equivalent to a regular (non-inline) user-defined function. A variable is a special type of components used to store an intermediate mapping result for further processing. You can also add a variable using the toolbar command.

Join (*Professional and Enterprise editions*)

The Join component allows you to join data in SQL and non-SQL modes. You can also add a Join component using the toolbar command.

Sort: Nodes/Rows

Inserts a component which allows you to sort nodes (see Sort Nodes/Rows[162]). You can also add a Sort component using the toolbar command.

Filter: Nodes/Rows

Inserts a Filter component that can filter data from any other component structure supported by MapForce, including databases. For more information, see Filters and Conditions[168]. You can also add a filter using the toolbar command.

SQL/NoSQL-WHERE/ORDER (*Professional and Enterprise editions*)

Inserts a component which allows you to filter database data conditionally. You can also access the SQL/NoSQL-WHERE/ORDER component via the toolbar command.

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 (e.g., month numbers to month names).

For more information, see Value-Maps [174]. You can also insert a Value-Map using the toolbar command.

IF-Else Condition

Inserts an If-Else Condition that is suitable for scenarios where you need to process a simple value conditionally. For more information, see Filters and Conditions [168]. You can also add an If-Else Condition using the toolbar command.

Exception (*Professional and Enterprise editions*)

The exception component allows you to interrupt a mapping process when a specific condition is met.

You can also add an Exception component using the toolbar command. In MapForce Enterprise Edition, this component also allows you to define Fault messages in WSDL mapping projects.

11.4 Component

This topic lists all the menu commands available in the **Component** menu.

Change Root Element

Allows you to change the root element of an XML instance document.

Edit Schema Definition in XMLSpy

To be able to edit a schema in Altova XMLSpy, you need to click an XML component and then select the option **Edit Schema Definition in XMLSpy**.

Edit FlexText Configuration (*Enterprise Edition*)

This command enables you to edit a FlexText file.

Add/Remove/Edit Database Objects (*Professional and Enterprise editions*)

Allows you to add, remove, and change database objects in a database component.

Create Mapping to EDI X12 997 (*Enterprise Edition*)

The X12 997 Functional Acknowledgment reports the status of the EDI interchange. All errors encountered during processing of the document are reported in it. MapForce can automatically generate a X12 997 document that you will be able to send to the recipient.

Create Mapping to EDI X12 999 (*Enterprise Edition*)

The X12 999 Implementation Acknowledgment Transaction Set reports HIPAA implementation guide non-compliance or application errors. MapForce can automatically generate an X12 999 component and automatically create the necessary mapping connections.

Refresh (*Professional and Enterprise editions*)

Reloads the structure of the currently active database component.

Add Duplicate Input Before/After

Inserts a copy of the selected item before/after this item. Duplicated input cannot be used as a data source. For more information, see Duplicate Input 72.

Remove Duplicate

Removes a duplicated item.

Comment/Processing Instructions

This option enables you to insert comments and processing instructions 115 into XML components.

Write Content as CDATA Section

This command creates a CDATA section 116 that is used to represent parts of a document as character data which would normally be interpreted as markup.

⊟ Database Table Actions (*Professional and Enterprise editions*)

Allows you to configure database insert, update, and delete actions, and other options for database records.

⊟ Query Database (*Professional and Enterprise editions*)

Creates a SELECT statement based on the table/field you clicked in the database component. Clicking a table/field makes this command active, and the SELECT statement is automatically placed into the **Select** window.

⊟ Align Tree Left

Makes the tree of a component left-justified.

⊟ Align Tree Right

Makes the tree of a component right-justified.

⊟ Properties

Displays the settings of the currently selected component. See Change Component Settings [71].

11.5 Connection

This topic lists all the menu commands available in the **Connection** menu.

⊟ Auto Connect Matching Children

Activates/deactivates the **Auto Connect Matching Children** option. For more information about connections and their types, see Connections 78.

⊟ Settings for Connect Matching Children

Helps you define matching-children connections. For details, see Matching-Children Connections 84.

⊟ Connect Matching Children

This command allows you to create multiple connections for items with the same names in source and target components. The settings you define in this dialog box apply if the (**Auto connect child items**) toolbar command has been enabled. For more information, see Matching-Children Connections 84.

⊟ Target Driven (Standard)

Changes the connector type to a standard mapping. For further information, see Target-driven vs. source-driven connections 81.

⊟ Copy-all (Copy Child Items)

Creates connections for all matching child items. The main benefit of copy-all connections is that they visually simplify the mapping workspace: One connection, represented by a thick line, is created instead of multiple connections. For details, see Copy-All Connections 86.

⊟ Source Driven (Mixed Content)

Changes the connection type to a source-driven connection that enables you to automatically map mixed content (text and child nodes) in the same order as in the XML *source* file. For more information, see Source-Driven Connections 82.

⊟ Properties

Opens the **Connection Settings** dialog box which allows you to define connection types and annotation settings. For more information, see Connection Settings 87.

11.6 Function

This topic lists all the menu commands available in the **Function** menu.

Create User-Defined Function

Creates a user-defined function [198] (UDF). You can also create a UDF using the toolbar command.

Create User-Defined Function from Selection

Creates a user-defined function based on the currently selected elements in the mapping window. For details, see Create UDFs [201]. You can also create a UDF from selection using the toolbar command.

Function Settings

Opens the **Edit User-defined Function** dialog box that allows you to change a UDF's settings. For details, see Edit UDFs [202].

Remove Function

Deletes the currently active user-defined function if you are working in a context which allows this.

Insert Input

Simple-Input components can be used as input parameters that are relevant to the entire mapping or only in the context of user-defined functions. For more information, see Simple Input [139] and Parameters in UDFs [204]. You can also insert a Simple-Input component using the toolbar command.

Insert Output

Simple-Output components can be used as output components in mappings and as output parameters of user-defined functions. For more information, see Simple Output [146] and Parameters in UDFs [204]. You can also insert a Simple-Output component using the toolbar button.

11.7 Output

This topic lists all the menu commands available in the **Output** menu.

XSLT 1.0/XSLT 2.0/XSLT 3.0/XQuery/Java/C#/C++/Built-In

Sets the transformation language in which the mapping should be executed. The selection of transformation languages depends on your MapForce edition. For details, see Transformation Languages [16]. You can also select transformation languages in the toolbar.

Validate Output File

Validates the output XML file against the referenced schema. See Validation [94].

Save Output File

Saves the data in the **Output** pane to a file.

Save All Output Files

Saves all the generated output files of dynamic mappings [417]. See also Tutorial 4 [56].

Regenerate Output

Reloads the data in the **Output** pane.

Run SQL/NoSQL-Script (*Professional and Enterprise editions*)

If an SQL/NoSQL script is currently visible in the **Output** pane, the script executes the mapping to the target database, taking the defined table actions into account.

Insert/Remove Bookmark

Inserts/removes a bookmark at the cursor position in the **Output** pane.

Next/Previous Bookmark

Navigates to the next/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 so that the document has a structured display: Each child node is offset from its parent by a single tab character. In the **Output** pane, the Tab size settings defined in the Text View Settings [96] dialog (Tabs group) take effect.

Text View Settings

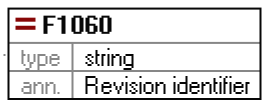
Displays the **Text View Settings** dialog box that allows you to customize text view settings in the **XQuery** pane (*Professional and Enterprise editions*), the **Output** pane, and the **XSLT** pane. The dialog also shows the currently defined hotkeys. For more information, see Text View Features [96].

11.8 View

This topic lists all the menu commands available in the **View** menu.

⊟ Show Annotations

Displays annotations in the component. You can also enable this option by clicking the toolbar button. If the **Show Types** icon is also active, both sets of information are shown in grid form (*see screenshot below*). You can also use annotations to label connections. For details, see Connection Settings 89.



⊟ Show Types

Displays data types in a component. You can also enable this option by clicking the toolbar button. If the **Show Annotations** icon is also active, then both sets of information are shown in grid form (*see Show Annotations above*).

⊟ Show Library in Function Header

Displays the library name in the function's header. You can also enable this option by clicking the toolbar button.

⊟ Show Tips

When you place the cursor over a function's header, you will see a tooltip summarizing what this function does. With the **Show Tips** option enabled, you can also see information about datatypes in a component.

⊟ XBRL Display Options (*Enterprise Edition*)

MapForce enables you to configure the following XBRL settings:

- The label language of XBRL items and their annotations
- The preferred label roles for XBRL item names
- The specific type of label roles of annotations for XBRL items
- Custom XBRL Taxonomy Packages

⊟ Show Selected Components Connectors/Connections from Source to Target

These options allow you to highlight connections selectively. To find out how these options work, see Connections 80.

⊟ Zoom

Opens the **Zoom** dialog box. You can enter the zoom factor numerically or drag the slider to change the zoom factor interactively.

⊟ Back/Forward

The **Back** and **Forward** commands allow you to switch between the previous and next mappings you have been working on, relative to the currently open mapping.

⊟ Status Bar

Switches on/off the **Status Bar** visible below the **Messages** window.

⊟ Libraries/Manage Libraries

Click **Libraries** to switch on/off the **Libraries** window. Click **Manage Libraries** to switch on/off the **Manage Libraries** window.

⊟ Messages

Switches on/off the Messages window[26]. When code is generated, the **Messages** window is automatically activated to show the validation result.

⊟ Overview

Switches on/off the Overview window[25]. Drag the rectangle to navigate your way through the mapping.

⊟ Project Window (*Professional and Enterprise editions*)

Switches on/off the **Project** window.

⊟ Debug Windows (*Professional and Enterprise editions*)

The debug mode enables you to analyze the context in which a particular value is produced. This information is available directly in the mapping and in the **Values**, **Context**, and **Breakpoints** windows.

11.9 Tools

This topic lists all the menu commands available in the **Tools** menu.

⊟ Global Resources

Opens the **Manage Global Resources** dialog box that enables you to add, edit, and delete settings applicable across multiple Altova applications. For more information, see Altova Global Resources [429].

⊟ Active Configuration

Allows you to select the currently active global resource configuration from a list of configurations. To create and configure different types of global resources, see Altova Global Resources [429].

⊟ Create Reversed Mapping

Creates a reversed mapping from the currently active mapping, which means the source component becomes the target component, and the target component becomes the source. Note that only direct connections between components are retained in the reversed mapping. It is likely that the new mapping will not be valid or suitable for preview in the **Output** pane. Therefore, the new mapping would require manual editing.

The following data is retained:

- Direct connections between components
- Direct connections between components in a chained mapping
- The type of connection [81]: Standard, Mixed-Content, Copy-All
- Pass-through component settings
- Database components (*Professional and Enterprise editions*)

The following data is *not* retained:

- Connections via functions, filters, etc.
- User-defined functions
- Web service components (*Enterprise Edition*)

⊟ XBRL Taxonomy Manager (*Enterprise Edition*)

XBRL Taxonomy Manager is a tool that allows you to install and manage XBRL taxonomies.

⊟ XML Schema Manager

XML Schema Manager is an Altova tool that provides a centralized way to install and manage XML schemas (DTDs for XML and XML Schemas) for use across all Altova's XBRL-enabled applications. For more information, see Schema Manager [122].

⊟ Customize

This option allows you to customize the MapForce graphical user interface. This includes showing/hiding toolbars as well as customizing menus [464] and keyboard shortcuts [465].

⊟ Restore Toolbars and Windows

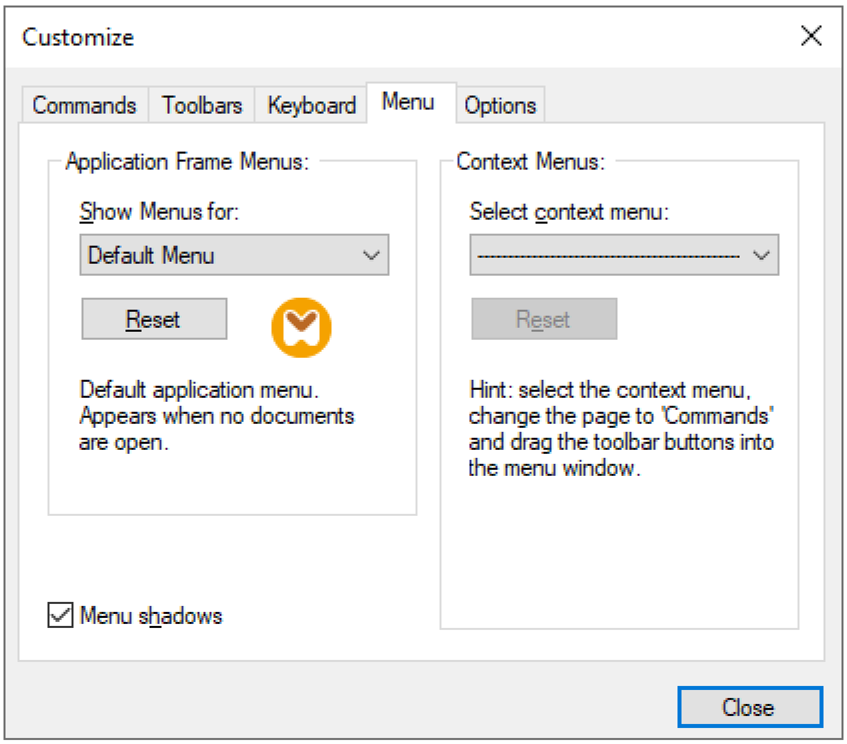
Resets the toolbars, entry helper windows, docked windows etc. to their defaults. You need to restart MapForce so that the changes take effect.

Options

Opens the **Options** dialog box that enables you to change the default MapForce settings. For more information, see Options 467.

11.9.1 Customize Menus

You can customize standard MapForce menus and context menus (e.g., to add, change, or remove commands). You can also revert your changes to the default state (**Reset**). To customize menus, go to **Tools | Customize** and click the **Menu** tab (*see screenshot below*).



Default Menu vs. MapForce Design
The *Default Menu* bar is displayed when no document is open in the main window. The *MapForce Design* menu bar is displayed when one or more mappings are open. Each menu bar can be customized separately. Customization changes made to one menu bar do not affect the other.

To customize a menu bar, select it from the *Show Menus For* drop-down list. Then click the **Commands** tab and drag commands from the *Commands* list box to the menu bar or into any of the menus.

Delete commands from menus
To delete an entire menu or a command inside a menu, do the following:

1. Select *Default Menu* or *MapForce Design* from the *Show Menus for* drop-down list.
2. With the **Customize** dialog open, select a toolbar command you want to delete or a command you want to delete from one of the menus.
3. Drag the toolbar command from the toolbar or the command from the menu. Alternatively, right-click the toolbar command or menu command and select **Delete**.

You can reset any menu bar to its default state by selecting it from the *Show Menus For* drop-down list and clicking the **Reset** button.

Customize context menus
Context menus appear when you right-click certain objects in the application's interface. Each of these context menus can be customized in the following way:

1. Select a context menu from the *Select context menu* drop-down list. This opens the respective context menu.
2. Open the **Commands** tab and drag a command from the *Commands* list box into the context menu.
3. To delete a command from the context menu, right-click that command and select **Delete**. Alternatively, drag the command out of the context menu.

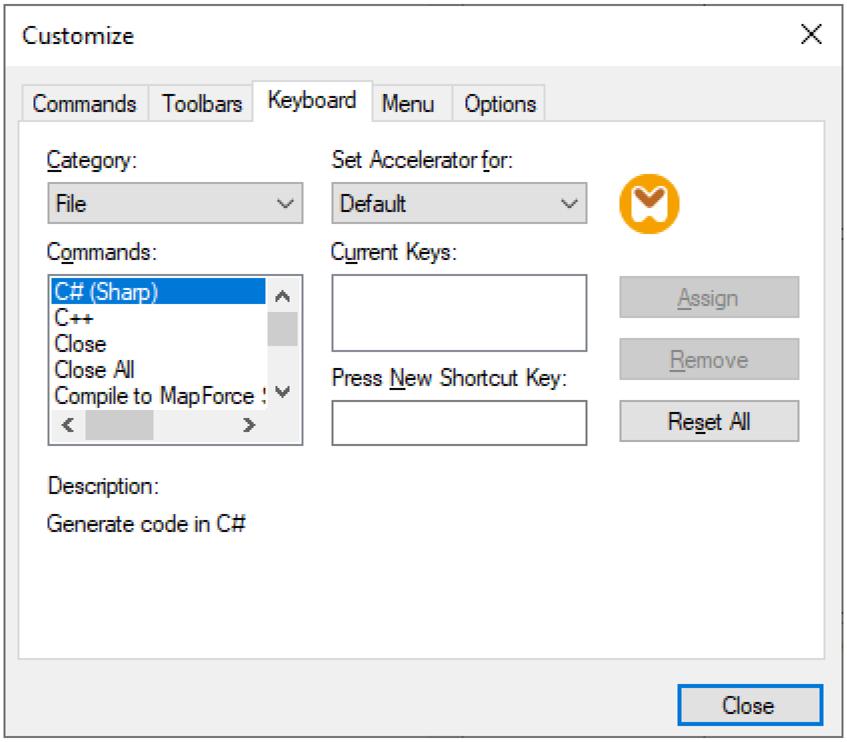
You can reset any context menu to its default state by selecting it in the *Select context menu* drop-down list and clicking the **Reset** button.

Menu shadows
Select the *Menu shadows* check box to give all menus shadows.

11.9.2 Customize Shortcuts

You can define or change keyboard shortcuts in MapForce as follows: Select the **Tools | Customize** and click the **Keyboard** tab. To assign a new shortcut to a command, take the following steps:

1. Select the **Tools | Customize** command and click the **Keyboard** tab (*see screenshot below*).
2. Click the *Category* combo box to select the menu name.
3. In the *Commands* list box, select the command you want to assign a new shortcut to.
4. Type in new shortcut keys in the *Press New Shortcut Key* text box and click **Assign**.



To clear the entry in the *Press New Shortcut Key* text box, press any of the control keys: **Ctrl**, **Alt** or **Shift**. To delete a shortcut, click the shortcut you want to delete in the *Current Keys* list box and click **Remove**.

Note: The *Set accelerator for* does not currently have any function.

Keyboard shortcuts

By default, MapForce provides the following keyboard shortcuts:

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
Arrow keys	
(up / down)	Select next item of component
Esc	Abandon edits/close dialog box
Return	Confirms a selection
Output window hotkeys	
CTRL + F2	Insert Remove/Bookmark
F2	Next Bookmark
SHIFT + F2	Previous Bookmark
CTRL + SHIFT + F2	Remove All Bookmarks
Zooming hotkeys	
CTRL + mouse wheel forward	Zoom In
CTRL + mouse wheel back	Zoom Out
CTRL + 0 (Zero)	Reset Zoom

11.9.3 Options

You can change general and other preferences in MapForce by selecting the **Tools | Options** command. The available options are described below.

General

In the *General* section, you can define the following options:

- *Show logo* | *Show on start:* Shows or hides an image (splash screen) when MapForce starts.
- *Mapping view.* You can enable/disable the gradient background in the **Mapping** pane (*Show gradient background*). You can also choose to limit annotation display to N lines. If the annotation text contains multiple lines, enabling this option shows only the first N lines (which is the value you specify) in the component. This setting also applies to SELECT statements visible in a component.
- *Default encoding for new components.*

Encoding name: The default encoding for new XML files can be set by selecting an option from the dropdown list. If a two- or four-byte encoding is selected as the default encoding (i.e. UTF-16, UCS-2, or UCS-4), you can also choose between little-endian and big-endian byte-ordering. This setting can also be changed individually for each component (see Change Component Settings 71).

Byte order: When a document with two-byte or four-byte character encoding is saved, the document can be saved either with little-endian or big-endian byte-ordering. You can also specify whether a byte order mark should be included.

- *Preview Settings:* The *Use execution timeout* option sets an execution timeout when you preview the mapping result in the **Output** pane.

- *On activating Output tab:* You can generate output to temporary files or write the output directly to an output file (*see below*).

  Generate output 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. For Professional and Enterprise editions: 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.

  Write directly to final output files: If the output file path contains folders that do not exist yet, an error will occur. This option overwrites any existing output files without requesting further confirmation.

- *Display text in steps of N million characters:* Specifies the maximum size of the text displayed in the **Output** pane when you preview mappings that generate large XML and text files. If the output text exceeds this value, you will need to click the **Load more** button to load the next chunk. For more information, see Preview and validate output 94.

Editing

In the *Editing* section, you can define mapping view options:

- *Align components on mouse dragging:* Specify whether components or functions should be aligned with other components, while you drag them in the Mapping window. For more information, see Align Components 72.

- *Smart component deletion:* MapForce allows you to keep connections even after deleting some transformation components 67. Keeping connections might be particularly useful with multiple child connections, because you will not have to restore every single child connection manually after deleting a transformation component. For details, see Keep Connections after Deleting Components 92.

Messages

The *Messages* section allows you to switch on message notifications such as suggesting connecting ancestor items, informing about the creation of multiple target components, and so on.

Generation (*Professional and Enterprise editions*)

The *Generation* section allows you to define settings for program code generation and MapForce Server Execution files.

⊟ Java

You may need to add a custom Java VM path, for example, if you are using a Java virtual machine which does not have an installer and does not create registry entries (e.g., Oracle's OpenJDK). You might also want to set this path if you need to override any Java VM path detected automatically by MapForce. For details, see Java 469.

⊟ XBRL (*Enterprise Edition*)

MapForce enables you to configure the following general (application-wide) XBRL settings:

- The label language of XBRL items and their annotations
- The preferred label roles for XBRL item names
- The specific type of label roles of annotations for XBRL items
- Custom XBRL Taxonomy Packages

⊟ Debugger (*Professional and Enterprise editions*)

In the *Debugger* section, you can define the following debugging settings:

- *Maximum storage length of values:* Defines the string length of values displayed in the **Values** window (at least 15 characters). Note that setting the storage length to a high value may deplete available system memory.

- *Keep full trace history:* Instructs MapForce to keep the history of all values processed by all connectors of all components in the mapping for the duration of debugging. If this option is enabled, all values processed since the beginning of debug execution will be stored in memory and available for your analysis in the **Values** window until you stop debugging. It is not recommended to enable this option if you are debugging data-intensive mappings, since it may slow down debugging execution and deplete available system memory. If this option is disabled, MapForce keeps only the most recent trace history for nodes related to the current execution position.

⊟ Database (*Professional and Enterprise editions*)

In the *Database* section, you can define database query settings.

⊟ Network Proxy

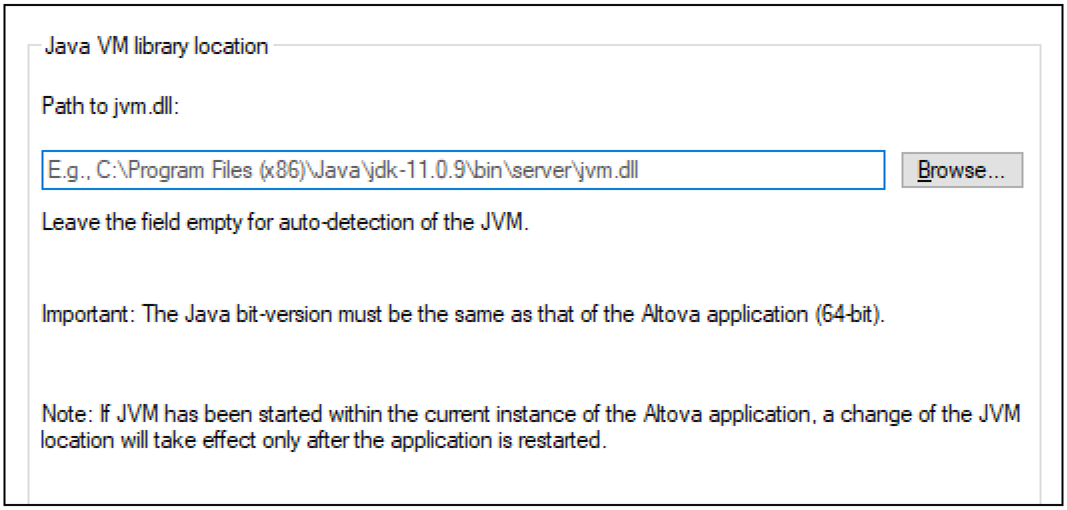
The *Network Proxy* section enables you to configure custom proxy settings. These settings affect how the application connects to the Internet. By default, the application uses the system's proxy settings, so you should not need to change the proxy settings in most cases. For more details, see Network Proxy 470.

11.9.3.1 Java

In the *Java* section (*see screenshot below*), you can optionally enter the path to a Java VM (Virtual Machine) on your file system. Note that adding a custom Java VM path is not always necessary. By default, MapForce attempts to detect the Java VM path automatically by reading (in this order) the Windows registry and the JAVA_HOME environment variable. The custom path added in this dialog box will take priority over any other

Java VM path detected automatically.

You may need to add a custom Java VM path, for example, if you are using a Java virtual machine which does not have an installer and does not create registry entries (e.g., Oracle's OpenJDK). You might also want to set this path if you need to override, for whatever reason, any Java VM path detected automatically by MapForce.



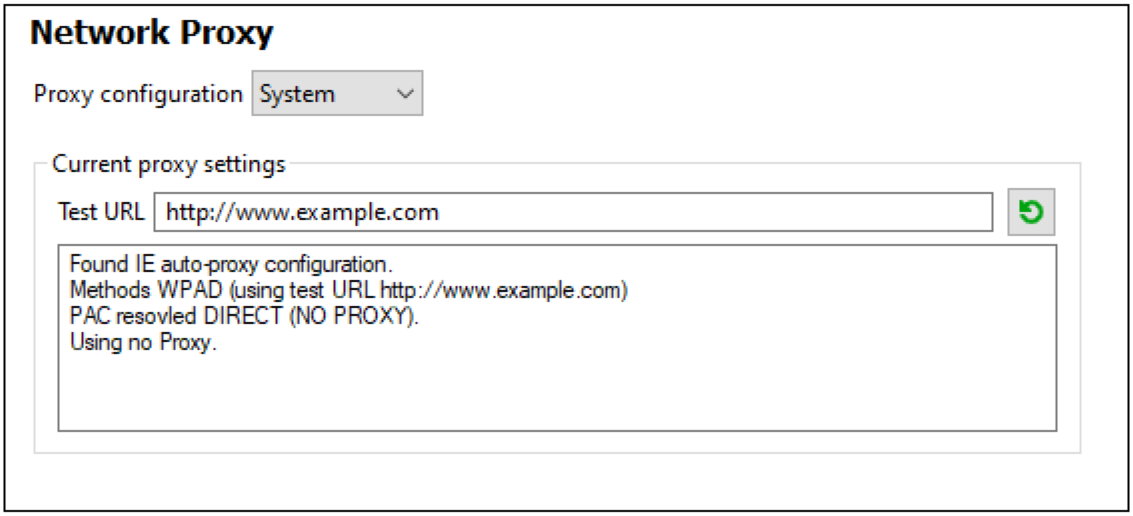
Note the following:

- The Java VM path is shared between Altova desktop (not server) applications. Consequently, if you change it in one application, it will automatically apply to all other Altova applications.
- The path must point to the `jvm.dll` file from the `\bin\server` or `\bin\client` directory, relative to the directory where the JDK was installed.
- The MapForce platform (32-bit, 64-bit) must be the same as that of the JDK.
- After changing the Java VM path, you may need to restart MapForce for the new settings to take effect.

11.9.3.2 Network Proxy

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 by selecting, in the *Proxy Configuration* combo box, either *Automatic* or *Manual* to configure the settings accordingly.

Note: The network proxy settings are shared among all Altova MissionKit applications. So, if you change the settings in one application, all MissionKit applications will be affected.

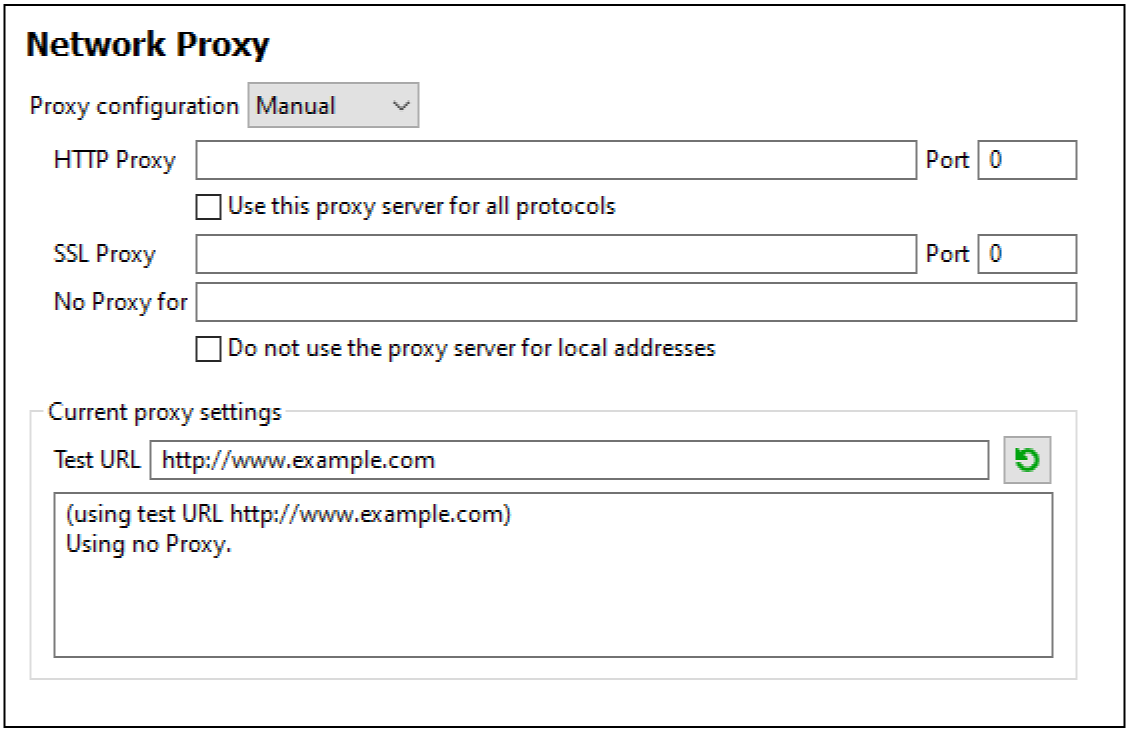


Use system proxy settings
Uses the Internet Explorer (IE) settings configurable via the system proxy settings. It also queries the settings configured with **`netsh.exe winhttp`**.

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:

- *HTTP Proxy:* Uses the specified host name and port for the HTTP protocol. If *Use this proxy server for all protocols* is selected, then the specified HTTP proxy is used for all protocols.
- *SSL Proxy:* Uses the specified host name and port for the SSL protocol.
- *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 (`.`).

Note: If a proxy server has been set and you want to deploy a mapping to Altova FlowForce Server, you must select the option *Do not use the proxy server for local addresses*.

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 *Authomatic proxy configuration*).

11.10 Window

This topic lists all the menu commands available in the **Window** menu.

⊟ Cascade

This command rearranges all open document windows so that they are all cascaded (i.e. staggered) on top of each other.

⊟ Tile Horizontal

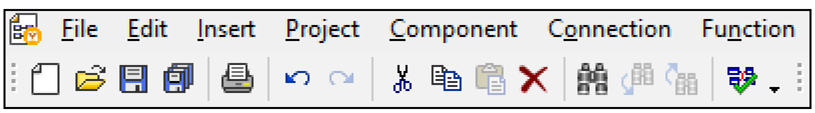
This command rearranges all open document windows as horizontal tiles, making them all visible at the same time.

⊟ Tile Vertical

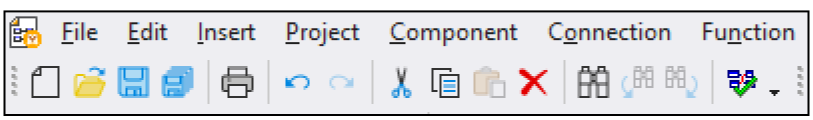
This command rearranges all open document windows as vertical tiles, making them all visible at the same time.

⊟ Classic/Light/Dark Theme

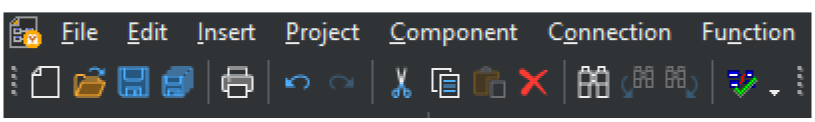
MapForce allows you to choose from the following themes: *Classic*, *Light*, and *Dark*. The examples of these themes are illustrated in the screenshots below. The default option is the Classic theme.



Classic Theme



Light Theme



Dark Theme

⊟ 1 <MappingName>

Refers to the first open mapping design. If there are more mappings opened at the same time, they will be listed in the context menu, too.

⊟ Windows

This list shows all currently open windows and enables you to quickly switch between them. You can also use the **Ctrl-TAB** or **CTRL F6** keyboard shortcuts to switch between the open windows.

11.11 Help

This topic lists all the menu commands available in the **Help** menu.

Table of Contents

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

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

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 Enter or List Topics. 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

License your product
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 license.*** 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 license. Enter your name, company, and e-mail address in the dialog and click **Request**. A license file is sent to the e-mail address you entered and should reach you in a few minutes. Save the license file to a suitable location.

  When you clicked **Request**, an entry field appeared at the bottom of the Request dialog. This field takes the path to the license file. Browse for or enter the path to the license file and click **OK**. (In the **Software Activation** dialog, you can also click **Upload a New License** to access a dialog in which the path to the license file is entered.) The software will be unlocked for a period of 30 days.

- ***Permanent license key.*** The **Software Activation** dialog allows you 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. Your license will be sent to you by e-mail in the form of a license file, which contains your license-data.

  There are three types of permanent license: *installed*, *concurrent user*, and *named user*. An installed license unlocks the software on a single computer. If you buy an installed license for *N* computers, then the license allows use of the software on up to *N* computers. A concurrent-user license for *N* concurrent users allows *N* users to run the software concurrently. (The software may be installed on 10N computers.) A named-user license authorizes a specific user to use the software on up to 5 different computers. To activate your software, click **Upload a New License**, and, in the dialog that appears, enter the path to the license file, and click **OK**.

Note: For multi-user licenses, each user will be prompted to enter his or her own name.

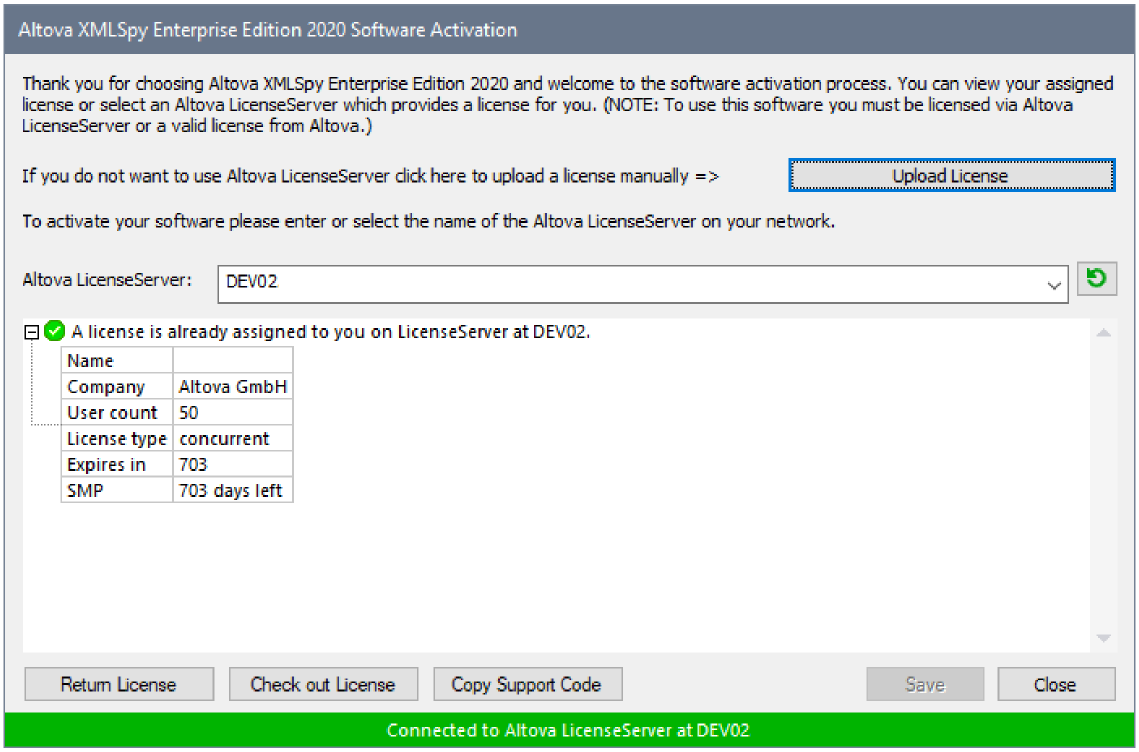
> *Your license email and the different ways to license (activate) your Altova product*
> The license email that you receive from Altova will contain your license file as an attachment. The license file has a **`.altova_licenses`** file extension.
>
> To activate your Altova product, you can do one of the following:
>
> - 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 a suitable location. In your Altova product, select the menu command **Help | Software Activation**, and then **Upload a New License**. Browse for or enter the path to the license file, and click **OK**.
> - 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 *(see below)* or (ii) assign the license to the product from Altova LicenseServer. *For more information about licensing via LicenseServer, read the rest of this topic.*

You can access the **Software Activation** dialog (*screenshot below*) at any time by clicking the **Help | Software Activation** command.

Activate your software
You can activate the software by registering the license in the Software Activation dialog or by licensing via Altova LicenseServer (*see details below*).

- *Registering the license in the Software Activation dialog*. In the dialog, click **Upload a New License** and browse for the license file. Click **OK** to confirm the path to the license file and to confirm any data you entered (your name in the case of multi-user licenses). Finish by clicking **Save**.

- *Licensing via Altova LicenseServer on your network:* To acquire a license via an Altova LicenseServer on your network, click **Use Altova LicenseServer**, located at the bottom of the **Software Activation** dialog. Select the machine on which the LicenseServer you want to use has been installed. Note that the auto-discovery of License Servers works by means of a broadcast sent out on the LAN. As these broadcasts are limited to a subnet, License Server must be on the same subnet as the client machine for auto-discovery to work. If auto-discovery does not work, then type in the name of the server. 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 (*see screenshot below showing the dialog in Altova XMLSpy*). Click **Save** to acquire the license.



After a machine-specific (aka installed) license has been acquired from LicenseServer, it cannot be returned to 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. After checking out a license, two things happen: (i) The **Software Activation** dialog will display the check-out information, including the time when the check-out period ends; (ii) 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 after the check-out period elapses, make sure that the check-out period you select adequately covers the period during which you will be working offline.

License check-ins must be to the same major version of the Altova product from which the license was checked out. So make sure to check in a license before you upgrade your Altova product to the next major version.

Note: For license check-outs to be possible, the check-out functionality must be enabled on LicenseServer. If this functionality has not been enabled, you will get an error message to this effect when you try to check out. 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

When you are ready to order a licensed version of the software product, you can use either the **Purchase a Permanent License Key** button in the **Software Activation** dialog (*see previous section*) or the **Order Form** command to proceed to the secure Altova Online Shop.

⊟ Registration

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

Checks with the Altova server whether a newer version than yours is currently available and displays a message accordingly.

⊟ Support Center

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

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

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

A link to the Altova website on the Internet. You can learn more about MapForce, related technologies and products on the Altova website.

MapForce Training

A link to the Online Training page on the Altova website. Here you can select from online courses conducted by Altova's expert trainers.

About MapForce

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.

12 Appendices

These appendices contain technical information about MapForce, its technical aspects and licensing. It also provides the list of key terms specific to MapForce and MapForce-related products. The section is organized into the following subsections:

- Support Notes 480
- Engine Information 482
- Technical Data 582
- License Information 584

12.1 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 11
- Windows Server 2008 R2 SP1 with Platform Update or newer

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

12.1.1 Supported Sources and Targets

When you change the transformation language of a MapForce mapping, certain features may not be supported for that specific language. The following table summarizes the compatibility of mapping formats and transformation languages in **MapForce Basic Edition**.

Remarks:

- *Built-in* means that you can execute the mapping by clicking the **Output** tab in MapForce or run it with MapForce Server.

12.1.2 Supported Features in Generated Code

The following table lists the features relevant to code generation and the extent of support in each language in **MapForce Basic Edition**.

Feature	XSLT 1.0	XSLT 2.0	XSLT 3.0
Supply parameters to the mapping [139]	●	●	●
Supply the input file names dynamically from the mapping [417]	●	●	●
Supply wildcard file names as mapping input [417] [1]		●	●
Generate the output file names dynamically from the mapping [417]		●	●
Return string values from the mapping [146]	●	●	●
Variables [150]		●	●
Sort components [162]		●	●
Grouping functions [184]		●	●
Filters [168]	●	●	●
Value-Map components [174]	●	●	●

Feature	XSLT 1.0	XSLT 2.0	XSLT 3.0
Dynamic node names [380]		●	●

Footnotes:

1. XSLT 2.0, XSLT 3.0, and XQuery use the `fn:collection` function. The implementation in the Altova XSLT 2.0, XSLT 3.0, and XQuery engines resolves wildcards. Other engines may behave differently.

12.2 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.

12.2.1 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 [482]
- XSLT 2.0 [482]
- XQuery 1.0 [484]

12.2.1.1 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 (` ` or ` `) or as an entity reference, ` ` .

12.2.1.2 XSLT 2.0

This section:

- Engine conformance [483]

- Backward compatibility 483
- Namespaces 483
- Schema awareness 484
- Implementation-specific behavior 484

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:`	`http://www.w3.org/2001/XMLSchema`
XPath 2.0 functions	`fn:`	`http://www.w3.org/2005/xpath-functions`

Typically, these namespaces will be declared on the `xsl:stylesheet` or `xsl:transform` element, as shown in the following listing:

```
<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:stylesheet>
```

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`.

12.2.1.3 XQuery 1.0

This section:

- Engine conformance 485
- Schema awareness 485
- Encoding 485
- Namespaces 483
- XML source and validation 486
- Static and dynamic type checking 486
- Library modules 486
- External functions 486
- Collations 486
- Precision of numeric data 487
- XQuery instructions support 487

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:`	`http://www.w3.org/2001/XMLSchema`
Schema instance	`xsi:`	`http://www.w3.org/2001/XMLSchema-instance`
Built-in functions	`fn:`	`http://www.w3.org/2005/xpath-functions`
Local functions	`local:`	`http://www.w3.org/2005/xquery-local-functions`

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 487. To use a specific collation, supply its URI as given in the list of supported collations 487. 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.

12.2.2 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 timezones 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.

12.2.2.1 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.

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**

XSLT functions 489

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 492
- Geolocation 509
- Image-related 521
- Numeric 525
- Sequence 547
- String 555
- Miscellaneous 561

12.2.2.1.1 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):	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

General 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 489]

XBRL functions

Altova XBRL functions can be used only with editions of Altova products that have XBRL support.

▼ xbrl-footnotes [altova:]

altova:xbrl-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 489]

12.2.2.1.2 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):	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

▼ Grouped by functionality

- Add a duration to xs:dateTime and return xs:dateTime 494
- Add a duration to xs:date and return xs:date 495
- Add a duration to xs:time and return xs:time 497
- Format and retrieve durations 496
- Remove timezone from functions that generate current date/time 498
- Return days, hours, minutes, and seconds from durations 499
- Return weekday as integer from date 501
- Return week number as integer from date 501
- Build date, time, or duration type from lexical components of each type 503
- Construct date, dateTime, or time type from string input 504
- Age-related functions 506
- Epoch time (Unix time) functions 507

Listed alphabetically

altova:add-days-to-date 495
altova:add-days-to-dateTime 494
altova:add-hours-to-dateTime 494
altova:add-hours-to-time 497
altova:add-minutes-to-dateTime 494
altova:add-minutes-to-time 497
altova:add-months-to-date 495
altova:add-months-to-dateTime 494
altova:add-seconds-to-dateTime 494
altova:add-seconds-to-time 497
altova:add-years-to-date 495
altova:add-years-to-dateTime 494
altova:age 506
altova:age-details 506
altova:build-date 503
altova:build-duration 503
altova:build-time 503
altova:current-dateTime-no-TZ 498
altova:current-date-no-TZ 498
altova:current-time-no-TZ 498
altova:date-no-TZ 498
altova:dateTime-from-epoch 507
altova:dateTime-from-epoch-no-TZ 507
altova:dateTime-no-TZ 498
altova:days-in-month 499
altova:epoch-from-dateTime 507
altova:hours-from-dateTimeDuration-accumulated 499
altova:minutes-from-dateTimeDuration-accumulated 499
altova:seconds-from-dateTimeDuration-accumulated 499
altova:format-duration 496
altova:parse-date 504
altova:parse-dateTime 504
altova:parse-duration 496
altova:parse-time 504
altova:time-no-TZ 498
altova:weekday-from-date 501
altova:weekday-from-dateTime 501
altova:weeknumber-from-date 502
altova:weeknumber-from-dateTime 502

[Top 492]

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 492]

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 492]

Format and retrieve durations XP3.1 XQ3.1

These functions parse an input **`xs:duration`** or **`xs:string`** and return, respectively, an **`xs:string`** or **`xs:duration`**.

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 492]

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*, **Minutes** *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 492]

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-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-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-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

▼ date-no-TZ [altova:]

altova:date-no-TZ(InputDate *as xs:date***) as xs:date** XP3.1 XQ3.1
This function takes an `xs:date` argument, removes the timezone part from it, and returns an `xs:date` value. Note that the date is not modified.

Examples

- **altova:date-no-TZ**(xs:date("2014-01-15+01:00")) returns 2014-01-15

dateTime-no-TZ [altova:]

altova:dateTime-no-TZ(InputDateTime *as xs:dateTime***) as xs:dateTime** XP3.1 XQ3.1
This function takes an `xs:dateTime` argument, removes the timezone part from it, and returns an `xs:dateTime` value. Note that neither the date nor the time is modified.

Examples

- **altova:dateTime-no-TZ**(xs:date("2014-01-15T14:00:00+01:00")) returns 2014-01-15T14:00:00

time-no-TZ [altova:]

altova:time-no-TZ(InputTime *as xs:time***) as xs:time** XP3.1 XQ3.1
This function takes an `xs:time` argument, removes the timezone part from it, and returns an `xs:time` value. Note that the time is not modified.

Examples

- **altova:time-no-TZ**(xs:time("14:00:00+01:00")) returns 14:00:00

[Top 492]

Return the number of days, hours, minutes, seconds from durations XP3.1 XQ3.1

These functions return the number of days in a month, and the number of hours, minutes, and seconds, respectively, from durations.

days-in-month [altova:]

altova:days-in-month(Year *as xs:integer***, Month** *as xs:integer***) as xs:integer** XP3.1 XQ3.1
Returns the number of days in the specified month. The month is specified by means of the `Year` and `Month` arguments.

Examples

- **altova:days-in-month**(2018, 10) returns 31
- **altova:days-in-month**(2018, 2) returns 28
- **altova:days-in-month**(2020, 2) returns 29

hours-from-dayTimeDuration-accumulated

altova:hours-from-dayTimeDuration-accumulated(DayAndTime *as xs:duration***) as xs:integer** XP3.1 XQ3.1
Returns the total number of hours in the duration submitted by the `DayAndTime` argument (which is of type `xs:duration`). The hours in the `Day` and `Time` components are added together to give a result that is an

integer. A new hour is counted only for a full 60 minutes. Negative durations result in a negative hour value.

Examples

- **altova:hours-from-dayTimeDuration-accumulated**(xs:duration("P5D")) returns 120, which is the total number of hours in 5 days.
- **altova:hours-from-dayTimeDuration-accumulated**(xs:duration("P5DT2H")) returns 122, which is the total number of hours in 5 days plus 2 hours.
- **altova:hours-from-dayTimeDuration-accumulated**(xs:duration("P5DT2H60M")) returns 123, which is the total number of hours in 5 days plus 2 hours and 60 mins.
- **altova:hours-from-dayTimeDuration-accumulated**(xs:duration("P5DT2H119M")) returns 123, which is the total number of hours in 5 days plus 2 hours and 119 mins.
- **altova:hours-from-dayTimeDuration-accumulated**(xs:duration("P5DT2H120M")) returns 124, which is the total number of hours in 5 days plus 2 hours and 120 mins.
- **altova:hours-from-dayTimeDuration-accumulated**(xs:duration("-P5DT2H")) returns -122

minutes-from-dayTimeDuration-accumulated

altova:minutes-from-dayTimeDuration-accumulated(DayAndTime *as xs:duration***) as xs:integer** XP3.1 XQ3.1

Returns the total number of minutes in the duration submitted by the `DayAndTime` argument (which is of type `xs:duration`). The minutes in the `Day` and `Time` components are added together to give a result that is an integer. Negative durations result in a negative minute value.

Examples

- **altova:minutes-from-dayTimeDuration-accumulated**(xs:duration("PT60M")) returns 60
- **altova:minutes-from-dayTimeDuration-accumulated**(xs:duration("PT1H")) returns 60, which is the total number of minutes in 1 hour.
- **altova:minutes-from-dayTimeDuration-accumulated**(xs:duration("PT1H40M")) returns 100
- **altova:minutes-from-dayTimeDuration-accumulated**(xs:duration("P1D")) returns 1440, which is the total number of minutes in 1 day.
- **altova:minutes-from-dayTimeDuration-accumulated**(xs:duration("-P1DT60M")) returns -1500

seconds-from-dayTimeDuration-accumulated

altova:seconds-from-dayTimeDuration-accumulated(DayAndTime *as xs:duration***) as xs:integer** XP3.1 XQ3.1

Returns the total number of seconds in the duration submitted by the `DayAndTime` argument (which is of type `xs:duration`). The seconds in the `Day` and `Time` components are added together to give a result that is an integer. Negative durations result in a negative seconds value.

Examples

- **altova:seconds-from-dayTimeDuration-accumulated**(xs:duration("PT1M")) returns 60, which is the total number of seconds in 1 minute.
- **altova:seconds-from-dayTimeDuration-accumulated**(xs:duration("PT1H")) returns 3600, which is the total number of seconds in 1 hour.
- **altova:seconds-from-dayTimeDuration-accumulated**(xs:duration("PT1H2M")) returns 3720
- **altova:seconds-from-dayTimeDuration-accumulated**(xs:duration("P1D")) returns 86400, which is the total number of seconds in 1 day.
- **altova:seconds-from-dayTimeDuration-accumulated**(xs:duration("-P1DT1M")) returns -

86460

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. 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. 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 492]

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 492]

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 492]

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.

D	Date
M	Month
Y	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.

D	Date
M	Month
Y	Year
H	Hour
m	minutes
s	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=[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.

H	Hour
m	minutes
s	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 492]

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 492]

Epoch time (Unix time) functions XP3.1 XQ3.1

Epoch time is a time system used on Unix systems. It defines any given point in time as being the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. Altova's Epoch time extension functions convert **xs:dateTime** values to Epoch time values and vice versa.

dateTime-from-epoch [altova:]

altova:dateTime-from-epoch(Epoch *as xs:decimal* **as xs:dateTime** XP3.1 XQ3.1

Epoch time is a time system used on Unix systems. It defines any given point in time as being the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. The **dateTime-from-epoch** function returns the **xs:dateTime** equivalent of an Epoch time, adjusts it for the local timezone, and includes the timezone information in the result.

The function takes an `xs:decimal` argument and returns an `xs:dateTime` value that includes a **TZ** (timezone) part. The result is obtained by calculating the UTC **dateTime** equivalet of the Epoch time, and adding to it the local timezone (taken from the system clock). For example, if the function is executed on a machine that has been set to be in a timezone of +01:00 (relative to UTC), then after the UTC **dateTime** equivalent has been calculated, one hour will be added to the result. The timezone information, which is an optional lexical part of the **xs:dateTime** result, is also reported in the **dateTime** result. Compare this result with that of **dateTime-from-epoch-no-TZ**, and also see the function **epoch-from-dateTime**.

Examples

The examples below assume a local timezone of UTC +01:00. Consequently, the UTC **dateTime** equivalent of the submitted Epoch time will be incremented by one hour. The timezone is reported in the result.

- **altova:dateTime-from-epoch**(34) returns 1970-01-01T01:00:34+01:00
- **altova:dateTime-from-epoch**(62) returns 1970-01-01T01:01:02+01:00

dateTime-from-epoch-no-TZ [altova:]

altova:dateTime-from-epoch-no-TZ(Epoch *as xs:decimal* **as xs:dateTime** XP3.1 XQ3.1

Epoch time is a time system used on Unix systems. It defines any given point in time as being the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. The **dateTime-from-epoch-no-TZ** function returns the **xs:dateTime** equivalent of an Epoch time, adjusts it for the local timezone, but does not include the timezone information in the result.

The function takes an `xs:decimal` argument and returns an `xs:dateTime` value that does not includes a **TZ** (timezone) part. The result is obtained by calculating the UTC **dateTime** equivalet of the Epoch time, and adding to it the local timezone (taken from the system clock). For example, if the function is executed on a machine that has been set to be in a timezone of +01:00 (relative to UTC), then after the UTC **dateTime** equivalent has been calculated, one hour will be added to the result. The timezone information, which is an optional lexical part of the **xs:dateTime** result, is not reported in the **dateTime** result. Compare this result with that of **dateTime-from-epoch**, and also see the function **epoch-from-dateTime**.

Examples

The examples below assume a local timezone of UTC +01:00. Consequently, the UTC **dateTime** equivalent of the submitted Epoch time will be incremented by one hour. The timezone is not reported in the result.

- **altova:dateTime-from-epoch**(34) returns 1970-01-01T01:00:34
- **altova:dateTime-from-epoch**(62) returns 1970-01-01T01:01:02

epoch-from-dateTime [altova:]

altova:epoch-from-dateTime(dateTimeValue *as xs:dateTime***) as xs:decimal** XP3.1 XQ3.1

Epoch time is a time system used on Unix systems. It defines any given point in time as being the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. The **epoch-from-dateTime** function returns the Epoch time equivalent of the **xs:dateTime** that is submitted as the argument of the function. Note that you might have to explicitly construct the **xs:dateTime** value. The submitted **xs:dateTime** value may or may not contain the optional **TZ** (timezone) part.

Whether the timezone part is submitted as part of the argument or not, the local timezone offset (taken from the system clock) is subtracted from the submitted **`dateTimeValue`** argument. This produces the equivalent UTC time, from which the equivalent Epoch time is calculated. For example, if the function is executed on a machine that has been set to be in a timezone of +01:00 (relative to UTC), then one hour is subtracted from the submitted **`dateTimeValue`** before the Epoch value is calculated. Also see the function **`dateTime-from-epoch`**.

Examples

The examples below assume a local timezone of UTC +01:00. Consequently, one hour will be subtracted from the submitted **`dateTime`** before the Epoch time is calculated.

- **`altova:epoch-from-dateTime`**`(xs:dateTime("1970-01-01T01:00:34+01:00"))` returns `34`
- **`altova:epoch-from-dateTime`**`(xs:dateTime("1970-01-01T01:00:34"))` returns `34`
- **`altova:epoch-from-dateTime`**`(xs:dateTime("2021-04-01T11:22:33"))` returns `1617272553`

[Top 492]

12.2.2.1.3 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.

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**

format-geolocation [altova:]

`altova:format-geolocation(Latitude` *`as xs:decimal`*, **`Longitude`** *`as xs:decimal`*, **`GeolocationOutputStringFormat`** *`as xs:integer`*) **`as xs:string`** **XP3.1** **XQ3.1**

Takes the latitude and longitude as the first two arguments, and outputs the geolocation as a string. The third argument, **`GeolocationOutputStringFormat`**, is the format of the geolocation output string; it uses integer values from `1` to `4` to identify the output string format (*see 'Geolocation output string formats' 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` (521) function and the Exif metadata's attributes can be used to supply the input strings.

Examples

- `altova:format-geolocation(33.33, -22.22, 4)` returns the `xs:string` `"33.33 -22.22"`
- `altova:format-geolocation(33.33, -22.22, 2)` returns the `xs:string` `"33.33N 22.22W"`
- `altova:format-geolocation(-33.33, 22.22, 2)` returns the `xs:string` `"33.33S 22.22E"`
- `altova:format-geolocation(33.33, -22.22, 1)` returns the `xs:string` `"33°19'48.00"S 22°13'12.00"E"`

Geolocation output string formats:

The supplied latitude and longitude is formatted in one of the output formats given below. The desired format is identified by its integer ID (`1 to 4`). Latitude values range from `+90` to `-90` (`N` to `S`). Longitude values range from `+180` to `-180` (`E` to `W`).

1
Degrees, minutes, decimal seconds, with suffixed orientation (`N/S, E/W`) **`D°M'S.SS"N/S  D°M'S.SS"E/W`** *Example*: **`33°55'11.11"N  22°44'66.66"W`**

2
Decimal degrees, with suffixed orientation (`N/S`, `E/W`) **`D.DDN/S  D.DDE/W`** *Example*: **`33.33N  22.22W`**

3
Degrees, minutes, decimal seconds, with prefixed sign (`+/-`); plus sign for (**`N/E`**) is optional **`+/-D°M'S.SS"  +/-D°M'S.SS"`** *Example*: **`33°55'11.11"  -22°44'66.66"`**

4
Decimal degrees, with prefixed sign (`+/-`); plus sign for (**`N/E`**) is optional **`+/-D.DD  +/-D.DD`** *Example*: **`33.33 -22.22`**

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**

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` 521 function and the Exif metadata's `@Geolocation` 521 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""") returns the sequence of two `xs:decimals` (48.8582222222222, 24.2945)
- **altova:parse-geolocation**('48°51''29.6"N 24°17''40.2"') returns the sequence of two `xs:decimals` (48.8582222222222, 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`, `E/W`)
  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/E**) 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`, `E/W`)
  `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/E`**) is optional
  `+/-D°M.MM'  +/-D°M.MM'`
  Example: **`+33°55.55'  -22°44.44'`**

- Decimal degrees, with suffixed orientation (`N/S`, `E/W`)
  `D.DDN/S  D.DDW/E`
  Example: **`33.33N  22.22W`**

- Decimal degrees, with prefixed sign (`+/-`); the plus sign for (**`N/S E/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` [521] function and the Exif metadata's `@Geolocation` [521] 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""")` 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`, `E/W`)
  `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/E`**) 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`, `E/W`)
  `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/E`**) is optional
  `+/-D°M.MM'  +/-D°M.MM'`
  Example: **`+33°55.55'  -22°44.44'`**

- Decimal degrees, with suffixed orientation (`N/S`, `E/W`)
  `D.DDN/S  D.DDW/E`
  Example: **`33.33N  22.22W`**

- Decimal degrees, with prefixed sign (`+/-`); the plus sign for (**`N/S E/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` 521 function and the Exif metadata's `@Geolocation` 521 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""")` 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`, `E/W`)
  `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/E`**) 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`, `E/W`)
  `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/E`**) is optional
  `+/-D°M.MM'  +/-D°M.MM'`
  Example: **`+33°55.55'  -22°44.44'`**

- Decimal degrees, with suffixed orientation (`N/S`, `E/W`)
  `D.DDN/S   D.DDW/E`
  Example: **`33.33N   22.22W`**

- Decimal degrees, with prefixed sign (`+/-`); the plus sign for (**`N/S E/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

geolocations-bounding-rectangle [altova:]

`altova:geolocations-bounding-rectangle(Geolocations` *`as xs:sequence`*, **`GeolocationOutputStringFormat`** *`as xs:integer`*`)` **`as xs:string`** **XP3.1** **XQ3.1**

Takes a sequence of strings as its first argument; each string in the sequence is a geolocation. The function returns a sequence of two strings which are, respectively, the top-left and bottom-right geolocation coordinates of a bounding rectangle that is optimally sized to enclose all the geolocations submitted in the first argument. The formats in which a geolocation input string can be supplied are listed below (*see 'Geolocation input string formats'*). Latitude values range from `+90` to `-90` (`N` to `S`). Longitude values range from `+180` to `-180` (`E` to `W`).

The function's second argument specifies the format of the two geolocation strings in the output sequence. The argument takes an integer value from `1` to `4`, where each value identifies a different geolocation string format (*see 'Geolocation output string formats' below*).

Note: The `image-exif-data` 521 function and the Exif metadata's attributes can be used to supply the input strings.

Examples

- **`altova:geolocations-bounding-rectangle`**`(("48.2143531 16.3707266", "51.50939 -0.11832"), 1)` returns the sequence `("51°30'33.804"N 0°7'5.952"W", "48°12'51.67116"N 16°22'14.61576"E")`
- **`altova:geolocations-bounding-rectangle`**`(("48.2143531 16.3707266", "51.50939 -`

`0.11832", "42.5584577 -70.8893334"), 4)` returns the sequence `("51.50939 -70.8893334", "42.5584577 16.3707266")`

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`, `E/W`)
  `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/E`**) 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`, `E/W`)
  `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/E`**) is optional
  `+/-D°M.MM'  +/-D°M.MM'`
  Example: **`+33°55.55'  -22°44.44'`**

- Decimal degrees, with suffixed orientation (`N/S`, `E/W`)
  `D.DDN/S  D.DDW/E`
  Example: **`33.33N  22.22W`**

- Decimal degrees, with prefixed sign (`+/-`); the plus sign for (**`N/S E/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`

Geolocation output string formats:

The supplied latitude and longitude is formatted in one of the output formats given below. The desired format is identified by its integer ID (`1 to 4`). Latitude values range from `+90` to `-90` (`N` to `S`). Longitude values range from `+180` to `-180` (`E` to `W`).

1
Degrees, minutes, decimal seconds, with suffixed orientation (`N/S, E/W`) **`D°M'S.SS"N/S  D°M'S.SS"E/W`** *Example*: **`33°55'11.11"N  22°44'66.66"W`**

2
Decimal degrees, with suffixed orientation (`N/S`, `E/W`) **`D.DDN/S  D.DDE/W`** *Example*: **`33.33N  22.22W`**

3
Degrees, minutes, decimal seconds, with prefixed sign (`+/-`); plus sign for (**`N/E`**) is optional **`+/-D°M'S.SS"  +/-D°M'S.SS"`** *Example*: **`33°55'11.11"  -22°44'66.66"`**

4
Decimal degrees, with prefixed sign (`+/-`); plus sign for (**`N/E`**) is optional **`+/-D.DD  +/-D.DD`** *Example*: **`33.33 -22.22`**

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` 521 function and the Exif metadata's `@Geolocation` 521 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"""))` 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`, `E/W`)
  `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/E`**) 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`, `E/W`)
  `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/E`**) is optional
  `+/-D°M.MM'  +/-D°M.MM'`
  Example: **`+33°55.55'  -22°44.44'`**

- Decimal degrees, with suffixed orientation (`N/S`, `E/W`)
  `D.DDN/S  D.DDW/E`
  Example: **`33.33N  22.22W`**

- Decimal degrees, with prefixed sign (`+/-`); the plus sign for (**`N/S E/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-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` 521 function and the Exif metadata's `@Geolocation` 521 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""") 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`, `E/W`)
  `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/E`**) 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`, `E/W`)
  `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/E`**) is optional
  `+/-D°M.MM'  +/-D°M.MM'`
  Example: **`+33°55.55'  -22°44.44'`**

- Decimal degrees, with suffixed orientation (`N/S`, `E/W`)
  `D.DDN/S  D.DDW/E`
  Example: **`33.33N  22.22W`**

- Decimal degrees, with prefixed sign (`+/-`); the plus sign for (**`N/S E/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 509]

12.2.2.1.4 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):	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

▾ 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

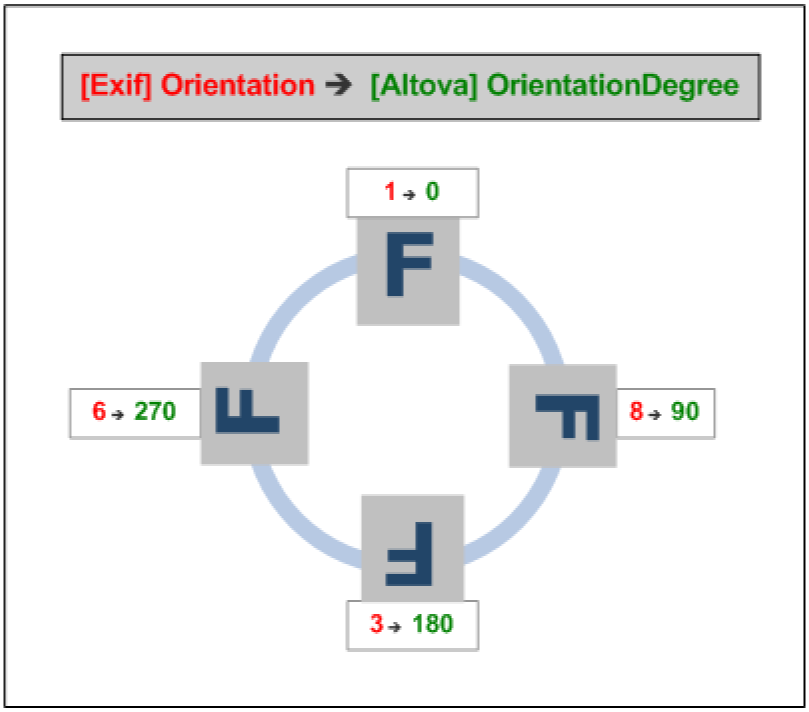
The Altova XPath/XQuery Engine generates the custom attribute **`Geolocation`** from standard Exif metadata tags. **`Geolocation`** is a concatenation of four Exif tags: `GPSLatitude`, `GPSLatitudeRef`, `GPSLongitude`, `GPSLongitudeRef`, with units added (*see table below*).

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
- SubjectLocation
- ExposureIndex
- SensingMethod
- FileSource
- SceneType
- CFAPattern
- CustomRendered
- ExposureMode
- WhiteBalance
- DigitalZoomRatio
- FocalLengthIn35mmFilm
- SceneCaptureType

- GainControl
- Contrast
- Saturation
- Sharpness
- DeviceSettingDescription
- SubjectDistanceRange
- ImageUniqueID

- GPSVersionID
- GPSLatitudeRef
- GPSLatitude
- GPSLongitudeRef
- GPSLongitude
- GPSAltitudeRef
- GPSAltitude
- GPSTimeStamp
- GPSSatellites
- GPSStatus
- GPSMeasureMode
- GPSDOP
- GPSSpeedRef
- GPSSpeed
- GPSTrackRef
- GPSTrack
- GPSImgDirectionRef
- GPSImgDirection
- GPSMapDatum
- GPSDestLatitudeRef
- GPSDestLatitude
- GPSDestLongitudeRef
- GPSDestLongitude
- GPSDestBearingRef
- GPSDestBearing
- GPSDestDistanceRef
- GPSDestDistance
- GPSProcessingMethod
- GPSAreaInformation
- GPSDateStamp
- GPSDifferential

[Top 521]

12.2.2.1.5 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 525]

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 525]

Number-formatting functions

[Top 525]

12.2.2.1.6 XPath/XQuery Functions: Schema

The Altova extension functions listed below return schema information. Given below are descriptions of the functions, together with (i) examples and (ii) a listing of schema components and their respective properties. They can be used with Altova's **XPath 3.0** and **XQuery 3.0** engines and are available in XPath/XQuery contexts.

Schema information from schema documents
The function **`altova:schema`** has two arguments: one with zero arguments and the other with two arguments. The zero-argument function returns the whole schema. You can then, from this starting point, navigate into the schema to locate the schema components you want. The two-argument function returns a specific component kind that is identified by its QName. In both cases, the return value is a function. To navigate into the returned component, you must select a property of that specific component. If the property is a non-atomic item (that is, if it is a component), then you can navigate further by selecting a property of this component. If the selected property is an atomic item, then the value of the item is returned and you cannot navigate any further.

Note: In XPath expressions, the schema must be imported into the processing environment (for example, into XSLT) with the **`xslt:import-schema`** instruction. In XQuery expressions, the schema must be explicitly imported using a schema import.

Schema information from XML nodes
The function **`altova:type`** submits the node of an XML document and returns the node's type information from the PSVI.

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

▼ Schema (zero arguments)

`altova:schema() as (function(xs:string) as item()*)?` XP3.1 XQ3.1
Returns the **schema** component as a whole. You can navigate further into the **schema** component by selecting one of the **schema** component's properties.

- If this property is a component, you can navigate another step deeper by selecting one of this component's properties. This step can be repeated to navigate further into the schema.
- If the component is an atomic value, the atomic value is returned and you cannot navigate any deeper.

The properties of the **schema** component are:

```
"type definitions"
"attribute declarations"
"element declarations"
"attribute group definitions"
"model group definitions"
"notation declarations"
"identity-constraint definitions"
```

The properties of all other component kinds (besides **schema**) are listed below.

Note: In XQuery expressions, the schema must be explicitly imported. In XPath expressions, the schema must have been imported into the processing environment, for example, into XSLT with the **xslt:import** instruction.

Examples

- `import schema "" at "C:\Test\ExpReport.xsd"; for $typedef in altova:schema() ("type definitions")`
  `return $typedef ("name")` returns the names of all simple types or complex types in the schema

- `import schema "" at "C:\Test\ExpReport.xsd";`
  `altova:schema() ("type definitions")[1]("name")` returns the name of the first of all simple types or complex types in the schema

Components and their properties

Assertion

Property name	Property type	Property value
kind	string	"Assertion"
test	XPath Property Record	

Attribute Declaration

Property name	Property type	Property value
kind	string	"Attribute Declaration"
name	string	Local name of the attribute
target namespace	string	Namespace URI of the attribute
type definition	Simple Type or Complex Type	
scope	A function with properties ("class":"Scope", "variety": "global" or "local", "parent": the containing Complex Type or Attribute Group)	

value constraint	If present, a function with properties ("class": "Value Constraint", "variety": "fixed" or "default", "value": atomic value, "lexical form": string. Note that the "value" property is not available for namespace-sensitive types	
inheritable	boolean	

Attribute Group Declaration

Property name	Property type	Property value
kind	string	"Attribute Group Definition"
name	string	Local name of the attribute group
target namespace	string	Namespace URI of the attribute group
attribute uses	Sequence of (Attribute Use)	
attribute wildcard	Optional Attribute Wildcard	

Attribute Use

Property name	Property type	Property value
kind	string	"Attribute Use"
required	boolean	true if the attribute is required, false if optional
value constraint	See Attribute Declaration	
inheritable	boolean	

Attribute Wildcard

Property name	Property type	Property value
kind	string	"Wildcard"
namespace constraint	function with properties ("class": "Namespace Constraint", "variety": "any"|"enumeration"|"not", "namespaces": sequence of xs:anyURI, "disallowed names": list containing QNames and/or the strings "defined" and "definedSiblings"	
process contents	string ("strict"|"lax"|"skip")	

Complex Type

Property name	Property type	Property value
kind	string	"Complex Type"

name	string	Local name of the type (empty if anonymous)
target namespace	string	Namespace URI of the type (empty if anonymous)
base type definition	Complex Type Definition	
final	Sequence of strings ("restriction"|"extension")	
context	Empty sequence (not implemented)	
derivation method	string ("restriction"|"extension")	
abstract	boolean	
attribute uses	Sequence of Attribute Use	
attribute wildcard	Optional Attribute Wildcard	
content type	function with properties: ("class":"Content Type", "variety":string ("element-only"|"empty"|"mixed"|"simple"), particle: optional Particle, "open content": function with properties ("class":"Open Content", "mode": string ("interleave"|"suffix"), "wildcard": Wildcard), "simple type definition": Simple Type)	
prohibited substitutions	Sequence of strings ("restriction"|"extension")	
assertions	Sequence of Assertion	

Element Declaration

Property name	**Property type**	**Property value**
kind	string	"Complex Type"
name	string	Local name of the type (empty if anonymous)
target namespace	string	Namespace URI of the type (empty if anonymous)
type definition	Simple Type or Complex Type	
type table	function with properties ("class":"Type Table", "alternatives": sequence of Type Alternative, "default type definition": Simple Type or Complex Type)	
scope	function with properties ("class":"Scope", "variety": ("global"|"local"), "parent": optional Complex Type)	
value constraint	see Attribute Declaration	

nillable	boolean	
identity-constraint definitions	Sequence of Identity Constraint	
substitution group affiliations	Sequence of Element Declaration	
substitution group exclusions	Sequence of strings ("restriction"|"extension")	
disallowed substitutions	Sequence of strings ("restriction"|"extension"|"substitution")	
abstract	boolean	

Element Wildcard

Property name	Property type	Property value
kind	string	"Wildcard"
namespace constraint	function with properties ("class": "Namespace Constraint", "variety": "any"|"enumeration"|"not", "namespaces": sequence of xs:anyURI, "disallowed names": list containing QNames and/or the strings "defined" and "definedSiblings"	
process contents	string ("strict"|"lax"|"skip")	

Facet

Property name	Property type	Property value
kind	string	The name of the facet, for example "minLength" or "enumeration"
value	depends on facet	The value of the facet
fixed	boolean	
typed-value	For the enumeration facet only, array(xs:anyAtomicType*)	An array containing the enumeration values, each of which may in general be a sequence of atomic values. (Note: for the enumeration facet, the "value" property is a sequence of strings, regardless of the actual type)

Identity Constraint

Property name	Property type	Property value
kind	string	"Identity-Constraint Definition"

name	string	Local name of the constraint
target namespace	string	Namespace URI of the constraint
identity-constraint category	string ("key"|"unique"|"keyRef")	
selector	XPath Property Record	
fields	Sequence of XPath Property Record	
referenced key	(For keyRef only): Identity Constraint	The corresponding key constraint

⊟ Model Group

Property name	Property type	Property value
kind	string	"Model Group"
compositor	string ("sequence"|"choice"|"all")	
particles	Sequence of Particle	

⊟ Model Group Definition

Property name	Property type	Property value
kind	string	"Model Group Definition"
name	string	Local name of the model group
target namespace	string	Namespace URI of the model group
model group	Model Group	

⊟ Notation

Property name	Property type	Property value
kind	string	"Notation Declaration"
name	string	Local name of the notation
target namespace	string	Namespace URI of the notation
system identifier	anyURI	
public identifier	string	

⊟ Particle

Property name	Property type	Property value
kind	string	"Particle"
min occurs	integer	
max occurs	integer, or string("unbounded")	
term	Element Declaration, Element Wildcard, or ModelGroup	

Simple Type

Property name	Property type	Property value
kind	string	"Simple Type Definition"
name	string	Local name of the type (empty if anonymous)
target namespace	string	Namespace URI of the type (empty if anonymous)
final	Sequence of string("restriction"|"extension"|"list"|"union")	
context	containing component	
base type definition	Simple Type	
facets	Sequence of Facet	
fundamental facets	Empty sequence (not implemented)	
variety	string ("atomic"|"list"|"union")	
primitive type definition	Simple Type	
item type definition	(for list types only) Simple Type	
member type definitions	(for union types only) Sequence of Simple Type	

Type Alternative

Property name	Property type	Property value
kind	string	"Type Alternative"
test	XPath Property Record	
type definition	Simple Type or Complex Type	

XPath Property Record

Property name	Property type	Property value
namespace bindings	Sequence of functions with properties ("prefix": string, "namespace": anyURI)	
default namespace	anyURI	
base URI	anyURI	The static base URI of the XPath expression
expression	string	The XPath expression as a string

Schema (two arguments)

altova:schema(ComponentKind *as xs:string*, **Name** *as xs:QName*) **as (function(xs:string) as item()*)?** XP3.1 XQ3.1
Returns the component kind that is specified in the first argument which has a name that is the same as the name supplied in the second argument. You can navigate further by selecting one of the component's properties.

- If this property is a component, you can navigate another step deeper by selecting one of this component's properties. This step can be repeated to navigate further into the schema.
- If the component is an atomic value, the atomic value is returned and you cannot navigate any deeper.

Note: In XQuery expressions, the schema must be explicitly imported. In XPath expressions, the schema must have been imported into the processing environment, for example, into XSLT with the **xslt:import** instruction.

Examples

- ```
  import schema "" at "C:\Test\ExpReport.xsd";
  altova:schema("element declaration", xs:QName("OrgChart"))("type definition")
  ("content type")("particles")[3]!.("term")("kind")
  ```
  returns the `kind` property of the term of the third `particles` component. This particles component is a descendant of the element declaration having a `QName` of `OrgChart`

- ```
  import schema "" at "C:\Test\ExpReport.xsd";
  let $typedef := altova:schema("type definition", xs:QName("emailType"))
  for $facet in $typedef ("facets")
  return [$facet ("kind"), $facet("value")]
  ```
  returns, for each **facet** of each **emailType** component, an array containing that facet's kind and value

Components and their properties

Assertion

Property name	Property type	Property value
kind	string	"Assertion"
test	XPath Property Record	

Attribute Declaration

Property name	Property type	Property value
kind	string	"Attribute Declaration"
name	string	Local name of the attribute
target namespace	string	Namespace URI of the attribute
type definition	Simple Type or Complex Type	
scope	A function with properties ("class":"Scope", "variety": "global" or "local", "parent": the containing	

	Complex Type or Attribute Group)	
value constraint	If present, a function with properties ("class": "Value Constraint", "variety": "fixed" or "default", "value": atomic value, "lexical form": string. Note that the "value" property is not available for namespace-sensitive types	
inheritable	boolean	

Attribute Group Declaration

Property name	Property type	Property value
kind	string	"Attribute Group Definition"
name	string	Local name of the attribute group
target namespace	string	Namespace URI of the attribute group
attribute uses	Sequence of (Attribute Use)	
attribute wildcard	Optional Attribute Wildcard	

Attribute Use

Property name	Property type	Property value
kind	string	"Attribute Use"
required	boolean	true if the attribute is required, false if optional
value constraint	See Attribute Declaration	
inheritable	boolean	

Attribute Wildcard

Property name	Property type	Property value
kind	string	"Wildcard"
namespace constraint	function with properties ("class": "Namespace Constraint", "variety": "any"|"enumeration"|"not", "namespaces": sequence of xs:anyURI, "disallowed names": list containing QNames and/or the strings "defined" and "definedSiblings"	
process contents	string ("strict"|"lax"|"skip")	

Complex Type

Property name	Property type	Property value

kind	string	"Complex Type"
name	string	Local name of the type (empty if anonymous)
target namespace	string	Namespace URI of the type (empty if anonymous)
base type definition	Complex Type Definition	
final	Sequence of strings ("restriction"|"extension")	
context	Empty sequence (not implemented)	
derivation method	string ("restriction"|"extension")	
abstract	boolean	
attribute uses	Sequence of Attribute Use	
attribute wildcard	Optional Attribute Wildcard	
content type	function with properties: ("class":"Content Type", "variety":string ("element-only"|"empty"|"mixed"|"simple"), particle: optional Particle, "open content": function with properties ("class":"Open Content", "mode": string ("interleave"|"suffix"), "wildcard": Wildcard), "simple type definition": Simple Type)	
prohibited substitutions	Sequence of strings ("restriction"|"extension")	
assertions	Sequence of Assertion	

Element Declaration

Property name	Property type	Property value
kind	string	"Complex Type"
name	string	Local name of the type (empty if anonymous)
target namespace	string	Namespace URI of the type (empty if anonymous)
type definition	Simple Type or Complex Type	
type table	function with properties ("class":"Type Table", "alternatives": sequence of Type Alternative, "default type definition": Simple Type or Complex Type)	
scope	function with properties ("class":"Scope", "variety": ("global"|"local"), "parent": optional Complex Type)	

value constraint	see Attribute Declaration	
nillable	boolean	
identity-constraint definitions	Sequence of Identity Constraint	
substitution group affiliations	Sequence of Element Declaration	
substitution group exclusions	Sequence of strings ("restriction"|"extension")	
disallowed substitutions	Sequence of strings ("restriction"|"extension"|"substitution")	
abstract	boolean	

Element Wildcard

Property name	Property type	Property value
kind	string	"Wildcard"
namespace constraint	function with properties ("class": "Namespace Constraint", "variety": "any"|"enumeration"|"not", "namespaces": sequence of xs:anyURI, "disallowed names": list containing QNames and/or the strings "defined" and "definedSiblings"	
process contents	string ("strict"|"lax"|"skip")	

Facet

Property name	Property type	Property value
kind	string	The name of the facet, for example "minLength" or "enumeration"
value	depends on facet	The value of the facet
fixed	boolean	
typed-value	For the enumeration facet only, array(xs:anyAtomicType*)	An array containing the enumeration values, each of which may in general be a sequence of atomic values. (Note: for the enumeration facet, the "value" property is a sequence of strings, regardless of the actual type)

Identity Constraint

Property name	Property type	Property value

kind	string	"Identity-Constraint Definition"
name	string	Local name of the constraint
target namespace	string	Namespace URI of the constraint
identity-constraint category	string ("key"|"unique"|"keyRef")	
selector	XPath Property Record	
fields	Sequence of XPath Property Record	
referenced key	(For keyRef only): Identity Constraint	The corresponding key constraint

Model Group

Property name	Property type	Property value
kind	string	"Model Group"
compositor	string ("sequence"|"choice"|"all")	
particles	Sequence of Particle	

Model Group Definition

Property name	Property type	Property value
kind	string	"Model Group Definition"
name	string	Local name of the model group
target namespace	string	Namespace URI of the model group
model group	Model Group	

Notation

Property name	Property type	Property value
kind	string	"Notation Declaration"
name	string	Local name of the notation
target namespace	string	Namespace URI of the notation
system identifier	anyURI	
public identifier	string	

Particle

Property name	Property type	Property value
kind	string	"Particle"
min occurs	integer	
max occurs	integer, or string("unbounded")	

term	Element Declaration, Element Wildcard, or ModelGroup	

Simple Type

Property name	Property type	Property value
kind	string	"Simple Type Definition"
name	string	Local name of the type (empty if anonymous)
target namespace	string	Namespace URI of the type (empty if anonymous)
final	Sequence of string("restriction"|"extension"|"list"|"union")	
context	containing component	
base type definition	Simple Type	
facets	Sequence of Facet	
fundamental facets	Empty sequence (not implemented)	
variety	string ("atomic"|"list"|"union")	
primitive type definition	Simple Type	
item type definition	(for list types only) Simple Type	
member type definitions	(for union types only) Sequence of Simple Type	

Type Alternative

Property name	Property type	Property value
kind	string	"Type Alternative"
test	XPath Property Record	
type definition	Simple Type or Complex Type	

XPath Property Record

Property name	Property type	Property value
namespace bindings	Sequence of functions with properties ("prefix": string, "namespace": anyURI)	
default namespace	anyURI	
base URI	anyURI	The static base URI of the XPath expression
expression	string	The XPath expression as a string

Type

altova:type(Node *as item?*) **as (function(xs:string) as item()*)?** XP3.1 XQ3.1

The function **altova:type** submits an element or attribute node of an XML document and returns the node's type information from the PSVI.

Note: The XML document must have a schema declaration so that the schema can be referenced.

Examples

- ```
  for $element in //Email
  let $type := altova:type($element)
  return $type
  ```
  returns a function that contains the `Email` node's type information

- ```
  for $element in //Email
  let $type := altova:type($element)
  return $type ("kind")
  ```
  takes the `Email` node's type component (Simple Type or Complex Type) and returns the value of the component's **kind** property

The **"_props"** parameter returns the properties of the selected component. For example:

- ```
  for $element in //Email
  let $type := altova:type($element)
  return ($type ("kind"), $type ("_props"))
  ```
  takes the `Email` node's type component (Simple Type or Complex Type) and returns (i) the value of the component's **kind** property, and then (ii) the properties of that component.

Components and their properties

Assertion

Property name	Property type	Property value
kind	string	"Assertion"
test	XPath Property Record	

Attribute Declaration

Property name	Property type	Property value
kind	string	"Attribute Declaration"
name	string	Local name of the attribute
target namespace	string	Namespace URI of the attribute
type definition	Simple Type or Complex Type	
scope	A function with properties ("class":"Scope", "variety": "global" or "local", "parent": the containing	
  ```

	Complex Type or Attribute Group)	
value constraint	If present, a function with properties ("class": "Value Constraint", "variety": "fixed" or "default", "value": atomic value, "lexical form": string. Note that the "value" property is not available for namespace-sensitive types	
inheritable	boolean	

Attribute Group Declaration

Property name	Property type	Property value
kind	string	"Attribute Group Definition"
name	string	Local name of the attribute group
target namespace	string	Namespace URI of the attribute group
attribute uses	Sequence of (Attribute Use)	
attribute wildcard	Optional Attribute Wildcard	

Attribute Use

Property name	Property type	Property value
kind	string	"Attribute Use"
required	boolean	true if the attribute is required, false if optional
value constraint	See Attribute Declaration	
inheritable	boolean	

Attribute Wildcard

Property name	Property type	Property value
kind	string	"Wildcard"
namespace constraint	function with properties ("class": "Namespace Constraint", "variety": "any"|"enumeration"|"not", "namespaces": sequence of xs:anyURI, "disallowed names": list containing QNames and/or the strings "defined" and "definedSiblings"	
process contents	string ("strict"|"lax"|"skip")	

Complex Type

Property name	Property type	Property value

kind	string	"Complex Type"
name	string	Local name of the type (empty if anonymous)
target namespace	string	Namespace URI of the type (empty if anonymous)
base type definition	Complex Type Definition	
final	Sequence of strings ("restriction"|"extension")	
context	Empty sequence (not implemented)	
derivation method	string ("restriction"|"extension")	
abstract	boolean	
attribute uses	Sequence of Attribute Use	
attribute wildcard	Optional Attribute Wildcard	
content type	function with properties: ("class":"Content Type", "variety":string ("element-only"|"empty"|"mixed"|"simple"), particle: optional Particle, "open content": function with properties ("class":"Open Content", "mode": string ("interleave"|"suffix"), "wildcard": Wildcard), "simple type definition": Simple Type)	
prohibited substitutions	Sequence of strings ("restriction"|"extension")	
assertions	Sequence of Assertion	

⊟ Element Declaration

Property name	Property type	Property value
kind	string	"Complex Type"
name	string	Local name of the type (empty if anonymous)
target namespace	string	Namespace URI of the type (empty if anonymous)
type definition	Simple Type or Complex Type	
type table	function with properties ("class":"Type Table", "alternatives": sequence of Type Alternative, "default type definition": Simple Type or Complex Type)	
scope	function with properties ("class":"Scope", "variety": ("global"|"local"), "parent": optional Complex Type)	

value constraint	see Attribute Declaration	
nillable	boolean	
identity-constraint definitions	Sequence of Identity Constraint	
substitution group affiliations	Sequence of Element Declaration	
substitution group exclusions	Sequence of strings ("restriction"|"extension")	
disallowed substitutions	Sequence of strings ("restriction"|"extension"|"substitution")	
abstract	boolean	

Element Wildcard

Property name	Property type	Property value
kind	string	"Wildcard"
namespace constraint	function with properties ("class": "Namespace Constraint", "variety": "any"|"enumeration"|"not", "namespaces": sequence of xs:anyURI, "disallowed names": list containing QNames and/or the strings "defined" and "definedSiblings"	
process contents	string ("strict"|"lax"|"skip")	

Facet

Property name	Property type	Property value
kind	string	The name of the facet, for example "minLength" or "enumeration"
value	depends on facet	The value of the facet
fixed	boolean	
typed-value	For the enumeration facet only, array(xs:anyAtomicType*)	An array containing the enumeration values, each of which may in general be a sequence of atomic values. (Note: for the enumeration facet, the "value" property is a sequence of strings, regardless of the actual type)

Identity Constraint

Property name	Property type	Property value

kind	string	"Identity-Constraint Definition"
name	string	Local name of the constraint
target namespace	string	Namespace URI of the constraint
identity-constraint category	string ("key"|"unique"|"keyRef")	
selector	XPath Property Record	
fields	Sequence of XPath Property Record	
referenced key	(For keyRef only): Identity Constraint	The corresponding key constraint

⊟ Model Group

Property name	Property type	Property value
kind	string	"Model Group"
compositor	string ("sequence"|"choice"|"all")	
particles	Sequence of Particle	

⊟ Model Group Definition

Property name	Property type	Property value
kind	string	"Model Group Definition"
name	string	Local name of the model group
target namespace	string	Namespace URI of the model group
model group	Model Group	

⊟ Notation

Property name	Property type	Property value
kind	string	"Notation Declaration"
name	string	Local name of the notation
target namespace	string	Namespace URI of the notation
system identifier	anyURI	
public identifier	string	

⊟ Particle

Property name	Property type	Property value
kind	string	"Particle"
min occurs	integer	
max occurs	integer, or string("unbounded")	

term	Element Declaration, Element Wildcard, or ModelGroup	

Simple Type

Property name	Property type	Property value
kind	string	"Simple Type Definition"
name	string	Local name of the type (empty if anonymous)
target namespace	string	Namespace URI of the type (empty if anonymous)
final	Sequence of string("restriction"|"extension"|"list"|"union")	
context	containing component	
base type definition	Simple Type	
facets	Sequence of Facet	
fundamental facets	Empty sequence (not implemented)	
variety	string ("atomic"|"list"|"union")	
primitive type definition	Simple Type	
item type definition	(for list types only) Simple Type	
member type definitions	(for union types only) Sequence of Simple Type	

Type Alternative

Property name	Property type	Property value
kind	string	"Type Alternative"
test	XPath Property Record	
type definition	Simple Type or Complex Type	

XPath Property Record

Property name	Property type	Property value
namespace bindings	Sequence of functions with properties ("prefix": string, "namespace": anyURI)	
default namespace	anyURI	
base URI	anyURI	The static base URI of the XPath expression
expression	string	The XPath expression as a string

12.2.2.1.7 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.

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

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:elements**("MyElem", "rip") returns MyElement()*
- **altova:elements**("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-availabe() 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-availabe() 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.*

for-each-attribute-pair [altova:]

`altova:for-each-attribute-pair(Seq1` *`as element()?`*, **`Seq2`** *`as element()?`*, **`Function`** *`as function()`***`)`** `as item()*` XP3.1 XQ3.1

The first two arguments identify two elements, the attributes of which are used to build attribute pairs, where one attribute of a pair is obtained from the first element and the other attribute is obtained from the second element. Attribute pairs are selected on the basis of having the same name, and the pairs are ordered alphabetically (on their names) into a set. If, for one attribute no corresponding attribute on the other element exists, then the pair is "disjoint", meaning that it consists of one member only. The function item (third argument `Function`) is applied separately to each pair in the sequence of pairs (joint and disjoint), resulting in an output that is a sequence of items.

Examples

- **`altova:for-each-attribute-pair`**`(/Example/Test-A, /Example/Test-B, function($a, $b){$a+b})` returns ...

  `(2, 4, 6)` if
  ```
  <Test-A att1="1" att2="2" att3="3" />
  <Test-B att1="1" att2="2" att3="3" />
  ```

  `(2, 4, 6)` if
  ```
  <Test-A att2="2" att1="1" att3="3" />
  <Test-B att3="3" att2="2" att1="1" />
  ```

  `(2, 6)` if
  ```
  <Test-A att4="4" att1="1" att3="3" />
  <Test-B att3="3" att2="2" att1="1" />
  ```

  Note: The result **`(2, 6)`** *is obtained by way of the following action:* **`(1+1, ()+2, 3+3, 4+())`**. *If one of the operands is the empty sequence, as in the case of items 2 and 4, then the result of the addition is an empty sequence.*

- **`altova:for-each-attribute-pair`**`(/Example/Test-A, /Example/Test-B, concat#2)` returns

...

```
(11, 22, 33) if
<Test-A att1="1" att2="2" att3="3" />
<Test-B att1="1" att2="2" att3="3" />

(11, 2, 33, 4) if
<Test-A att4="4" att1="1" att3="3" />
<Test-B att3="3" att2="2" att1="1" />
```

▼ for-each-combination [altova:]

altova:for-each-combination(FirstSequence *as item()**, **SecondSequence** *as item()**, **Function($i,$j){$i || $j}) as item()*** XP3.1 XQ3.1

The items of the two sequences in the first two arguments are combined so that each item of the first sequence is combined, in order, once with each item of the second sequence. The function given as the third argument is applied to each combination in the resulting sequence, resulting in an output that is a sequence of items (*see example*).

⊟ *Examples*

- `altova:for-each-combination( ('a', 'b', 'c'), ('1', '2', '3'), function($i, $j) {$i || $j} )` returns `('a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3')`

▼ for-each-matching-attribute-pair [altova:]

altova:for-each-matching-attribute-pair(Seq1 *as element()?*, **Seq2** *as element()?*, **Function** *as function()*) **as item()*** XP3.1 XQ3.1

The first two arguments identify two elements, the attributes of which are used to build attribute pairs, where one attribute of a pair is obtained from the first element and the other attribute is obtained from the second element. Attribute pairs are selected on the basis of having the same name, and the pairs are ordered alphabetically (on their names) into a set. If, for one attribute no corresponding attribute on the other element exists, then no pair is built. The function item (third argument `Function`) is applied separately to each pair in the sequence of pairs, resulting in an output that is a sequence of items.

⊟ *Examples*

- `altova:for-each-matching-attribute-pair(/Example/Test-A, /Example/Test-B, function($a, $b){$a+b})` returns ...

```
(2, 4, 6) if
<Test-A att1="1" att2="2" att3="3" />
<Test-B att1="1" att2="2" att3="3" />

(2, 4, 6) if
<Test-A att2="2" att1="1" att3="3" />
<Test-B att3="3" att2="2" att1="1" />

(2, 6) if
<Test-A att4="4" att1="1" att3="3" />
<Test-B att3="3" att2="2" att3="1" />
```

- `altova:for-each-matching-attribute-pair(/Example/Test-A, /Example/Test-B, concat#2)` returns ...

  `(11, 22, 33)` if
  ```
  <Test-A att1="1" att2="2" att3="3" />
  <Test-B att1="1" att2="2" att3="3" />
  ```

  `(11, 33)` if
  ```
  <Test-A att4="4" att1="1" att3="3" />
  <Test-B att3="3" att2="2" att1="1" />
  ```

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)`

12.2.2.1.8 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.

create-hash-from-string[altova:]

altova:create-hash-from-string(InputString *as xs:string***) as xs:string** XP2 XQ1 XP3.1 XQ3.1
altova:create-hash-from-string(InputString *as xs:string*, **HashAlgo** *as xs:string***) as xs:string** XP2 XQ1 XP3.1 XQ3.1
Generates a hash string from `InputString` by using the hashing algorithm specified by the `HashAlgo` argument. The following hashing algorithms may be specified (in upper or lower case): **MD5**, **SHA-1**, **SHA-224**, **SHA-256**, **SHA-384**, **SHA-512**. If the second argument is not specified (*see the first signature above*), then the **SHA-256** hashing algorithm is used.

Examples

- `altova:create-hash-from-string('abc')` returns a hash string generated by using the **SHA-256** hashing algorithm.
- `altova:create-hash-from-string('abc', 'md5')` returns a hash string generated by using the **MD5** hashing algorithm.
- `altova:create-hash-from-string('abc', 'MD5')` returns a hash string generated by using the **MD5** hashing algorithm.

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`

format-string [altova:]

altova:format-string(InputString *as xs:string*, **FormatSequence** *as item()****) as xs:string** XP3.1 XQ3.1
The input string (first argument) contains positional parameters (`%1`, `%2`, etc). Each parameter is replaced by the string item that is located at the corresponding position in the format sequence (submitted as the second argument). So the first item in the format sequence replaces the positional parameter `%1`, the second item replaces `%2`, and so on. The function returns this formatted string that contains the replacements. If no string exists for a positional parameter, then the positional parameter itself is returned. This happens when the index of a positional parameter is greater than the number of items in the format sequence.

Examples

- **altova:format-string**('Hello %1, %2, %3', ('Jane','John','Joe')) returns "Hello Jane, John, Joe"
- **altova:format-string**('Hello %1, %2, %3', ('Jane','John','Joe', 'Tom')) returns "Hello Jane, John, Joe"
- **altova:format-string**('Hello %1, %2, %4', ('Jane','John','Joe', 'Tom')) returns "Hello Jane, John, Tom"
- **altova:format-string**('Hello %1, %2, %4', ('Jane','John','Joe')) returns "Hello Jane, John, %4"

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"`

12.2.2.1.9 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.

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

decode-string [altova:]

altova:decode-string(Input *as xs:base64Binary***) as xs:string** XP3.1 XQ3.1
altova:decode-string(Input *as xs:base64Binary*, **Encoding** *as xs:string***) as xs:string** XP3.1 XQ3.1

Decodes the submitted base64Binary input to a string using the specified encoding. If no encoding is specified, then the UTF-8 encoding is used. The following encodings are supported: `US-ASCII, ISO-8859-1, UTF-16, UTF-16LE, UTF-16BE, ISO-10646-UCS2, UTF-32, UTF-32LE, UTF-32BE, ISO-10646-UCS4`

Examples

- `altova:decode-string($XML1/MailData/Meta/b64B)` returns the base64Binary input as a UTF-8 encoded string
- `altova:decode-string($XML1/MailData/Meta/b64B, "UTF-8")` returns the base64Binary input as a UTF-8-encoded string
- `altova:decode-string($XML1/MailData/Meta/b64B, "ISO-8859-1")` returns the base64Binary input as an ISO-8859-1-encoded string

encode-string [altova:]

altova:encode-string(InputString *as xs:string***) as xs:base64Binaryinteger** XP3.1 XQ3.1
altova:encode-string(InputString *as xs:string*, **Encoding** *as xs:string***) as xs:base64Binaryinteger** XP3.1 XQ3.1

Encodes the submitted string using, if one is given, the specified encoding. If no encoding is given, then the UTF-8 encoding is used. The encoded string is converted to base64Binary characters, and the converted base64Binary value is returned. Initially, UTF-8 encoding is supported, and support will be extended to the following encodings: `US-ASCII, ISO-8859-1, UTF-16, UTF-16LE, UTF-16BE, ISO-10646-UCS2, UTF-32, UTF-32LE, UTF-32BE, ISO-10646-UCS4`

Examples

- `altova:encode-string("Altova")` returns the base64Binary equivalent of the UTF-8 encoded string `"Altova"`
- `altova:encode-string("Altova", "UTF-8")` returns the base64Binary equivalent of the UTF-8 encoded string `"Altova"`

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`

high-res-timer [altova:]

altova:high-res-timer() as xs:double XP3.1 XQ3.1

Returns a system high-resolution timer value in seconds. A high-resolution timer, when present on a system, enables high precision time measurements when these are required (for example, in animations and for determining precise code-execution time). This function provides the resolution of the system's high-res timer.

Examples

- **altova:high-res-timer**() returns something like `'1.16766146154566E6'`

parse-html [altova:]

altova:parse-html(HTMLText *as xs:string*) **as node()** XP3.1 XQ3.1

The `HTMLText` argument is a string that contains the text of an HTML document. The function creates an HTML tree from the string. The submitted string may or may not contain the HTML element. In either case, the root element of the tree is an element named **HTML**. It is best to make sure that the HTML code in the submitted string is valid HTML.

Examples

- **altova:parse-html**(`"<html><head/><body><h1>Header</h1></body></html>"`) creates an HTML tree from the submitted string

sleep[altova:]

altova:sleep(Millisecs *as xs:integer*) **as empty-sequence()** XP2 XQ1 XP3.1 XQ3.1

Suspends execution of the current operation for the number of milliseconds given by the `Millisecs` argument.

Examples

- **altova:sleep**(1000) suspends execution of the current operation for 1000 milliseconds.

[Top 561]

12.2.2.2 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 [564] and .NET [573], as well as MSXSL scripts for XSLT [579]. 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 [564]
- .NET Extension Functions [573]
- MSXSL Scripts for XSLT [579]

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.

12.2.2.2.1 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 [569]
- Java: Static Methods and Static Fields [570]
- Java: Instance Methods and Instance Fields [571]
- Datatypes: XPath/XQuery to Java [571]
- Datatypes: Java to XPath/XQuery [572]

Note the following

- If you are using an Altova desktop product, the Altova application attempts to detect the path to the Java virtual machine automatically, by reading (in this order): (i) the Windows registry, and (ii) the **`JAVA_HOME`** environment variable. You can also add a custom path in the Options dialog of the application; this entry will take priority over any other Java VM path detected automatically.
- If you are running an Altova server product on a Windows machine, the path to the Java virtual machine will be read first from the Windows registry; if this is not successful the **`JAVA_HOME`** environment variable will be used.

- If you are running an Altova server product on a Linux or macOS machine, then make sure that the **`JAVA_HOME`** environment variable is properly set and that the Java Virtual Machines library (on Windows, the **`jvm.dll`** file) can be located in either the **`\bin\server`** or **`\bin\client`** directory.

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 Files566 and User-Defined Jar Files568. Note that paths to class files not in the current directory and to all JAR files must be specified.

12.2.2.2.1.1 User-Defined Class Files

If access is via a class file, then there are four 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* 566.)
- The class file is not packaged. The XSLT or XQuery file is in the same folder as the class file. (*See example below* 567.)
- The class file is in a package. The XSLT or XQuery file is at some random location. (*See example below* 567.)
- The class file is not packaged. The XSLT or XQuery file is at some random location. (*See example below* 568.)

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 referenced, XSLT/XQuery file in same folder as class file

The example below calls the `getVehicleType()`method of the `Car` class. Let us say that: (i) the `Car` class file is in the following folder: `JavaProject/com/altova/extfunc`, and (ii) that this folder is the current folder in the example below. 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 referenced, XSLT/XQuery file at any location

The example below calls the `getCarColor()`method of the `Car` class. Let us say that the `Car` class file is in the folder `C:/JavaProject/com/altova/extfunc`, and the XSLT file is at any location. The location of the class file must then be 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.

12.2.2.2.1.2 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.

12.2.2.2.1.3 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 572, 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* 571):
  ```
  <xsl:value-of select="date:toString(date:new())" xmlns:date="java:java.util.Date" />
  ```

12.2.2.2.1.4 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>
```

12.2.2.2.1.5 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:jlang="java:java.lang">
  <xsl:param name="CurrentDate" select="date:new()"/>
  <xsl:template match="/">
      <enrollment institution-id="Altova School"
                date="{date:toString($CurrentDate)}"
                type="{jlang:Object.toString(jlang: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 `jlang.Object.getClass` method.
3. The object obtained by the `getClass` method is passed as the argument to the `jlang.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`.

12.2.2.2.1.6 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.

`xs:string`	`java.lang.String`
`xs:boolean`	`boolean` (primitive), `java.lang.Boolean`
`xs:integer`	int, long, short, byte, float, double, and the wrapper classes of these, such as `java.lang.Integer`
`xs:float`	`float` (primitive), `java.lang.Float`, `double` (primitive)
`xs:double`	`double` (primitive), `java.lang.Double`
`xs:decimal`	`float` (primitive), `java.lang.Float`, `double`(primitive), `java.lang.Double`

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.

12.2.2.2.1.7 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 `xsd: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).

12.2.2.2.2 .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 575
- .NET: Static Methods and Static Fields 576
- .NET: Instance Methods and Instance Fields 576
- Datatypes: XPath/XQuery to .NET 577
- Datatypes: .NET to XPath/XQuery 578

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:

`asm`	The name of the assembly to be loaded.
`ver`	The version number (maximum of four integers separated by periods).
`sn`	The key token of the assembly's strong name (16 hex digits).
`from`	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.
`partialname`	The partial name of the assembly. It is supplied to `Assembly.LoadWith.PartialName()`, which will attempt to load the assembly. If `partialname` is present, any other parameter is ignored.

`loc`	The locale, for example, `en-US`. The default is `neutral`.

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:

1. When the assembly is loaded from the GAC:
   ```
   declare namespace cs="clitype:MyManagedDLL.testClass?asm=MyManagedDLL;
         ver=1.2.3.4;loc=neutral;sn=b9f091b72dccfba8";
   ```

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.

12.2.2.2.2.1 .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 572, 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* 571):
  ```
  <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(date:get_Month(date:new(2008, 4, 29)))"
  xmlns:date="clitype:System.DateTime" />
  ```

12.2.2.2.2.2 .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>
```

12.2.2.2.2.3 .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.

12.2.2.2.2.4 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.

`xs:string`	`StringValue, string`
`xs:boolean`	`BooleanValue, bool`
`xs:integer`	`IntegerValue, decimal, long, integer, short, byte, double, float`
`xs:float`	`FloatValue, float, double`
`xs:double`	`DoubleValue, double`
`xs:decimal`	`DecimalValue, decimal, double, float`

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.

12.2.2.2.2.5 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).

12.2.2.2.3 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>
          <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.

12.3 Technical Data

This section contains information on some technical aspects of your software. This information is organized into the following sections:

- OS and Memory Requirements 582
- Altova Engines 582
- Unicode Support 583
- Internet Usage 583

12.3.1 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 11
- 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. As a result, 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.

12.3.2 Altova Engines

XML Validator

When opening an XML document, the application uses its built-in XML validator to check for well-formedness, to validate the document against a schema (if specified), and to 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 specifications. 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.

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. If one of these engines is included in the product, then documentation about implementation-specific behavior for each engine is given in the appendices of the documentation.

Note: Altova MapForce generates code using the XSLT 1.0, 2.0 and XQuery 1.0 engines.

12.3.3 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.

12.3.4 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.

12.4 License Information

This section contains information about:

- the distribution of this software product
- software activation and license metering
- the license agreement governing the use of this product

Please read this information carefully. It is binding upon you since you agreed to these terms when you installed this software product.

To view the terms of any Altova license, go to the Altova Legal Information page at the Altova website.

12.4.1 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 for 30 days before making a purchasing decision. *(Note: Altova MobileTogether Designer is licensed free of charge.)*
- Once you decide to buy the software, you can place your order online at the Altova website and 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 an onscreen help system that can be accessed from within the application interface. The latest version of the user manual is available at www.altova.com in (i) HTML format for online browsing, and (ii) 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 the 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 must purchase a product license, which is delivered in the form of a license file containing a key code. Unlock the product by uploading the license file in the Software Activation dialog of your product.

You can purchase product licenses at https://shop.altova.com/.

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 distribute only the installer file, provided that this file is not modified in any way. Any person who 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.

12.4.2 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-user 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 Service Name Registry 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.

Note about certificates

Your Altova application contacts the Altova licensing server (`link.altova.com`) via HTTPS. For this communication, Altova uses a registered SSL certificate. If this certificate is replaced (for example, by your IT department or an external agency), then your Altova application will warn you about the connection being insecure. You could use the replacement certificate to start your Altova application, but you would be doing this at your own risk. If you see a *Non-secure connection* warning message, check the origin of the certificate and consult your IT team (who would be able to decide whether the interception and replacement of the Altova certificate should continue or not).

If your organization needs to use its own certificate (for example, to monitor communication to and from client machines), then we recommend that you install Altova's free license management software, Altova LicenseServer, on your network. Under this setup, client machines can continue to use your organization's certificates, while Altova LicenseServer can be allowed to use the Altova certificate for communication with Altova.

12.4.3 Altova End-User License Agreement

- The Altova End-User License Agreement is available here: https://www.altova.com/legal/eula
- Altova's Privacy Policy is available here: https://www.altova.com/privacy

Index

.

.NET extension functions,
constructors, 575
datatype conversions (.NET to XPath/XQuery), 578
datatype conversions (XPath/XQuery to .NET), 577
for XSLT and XQuery, 573
instance methods, instance fields, 576
overview, 573
static methods, static fields, 576

A

A to Z,
sort component, 162
abs,
as MapForce function (in xpath2 | numeric functions), 336
Actions,
connection-related, 78
add,
as MapForce function (in core | math functions), 257
Altova extensions,
chart functions (see chart functions), 488
Altova XML Parser,
about, 582
ATTLIST,
DTD namespace URIs, 106
auto-number,
as MapForce function (in core | generator functions), 249
avg,
as MapForce function (in core | aggregate functions), 228

B

Background Information, 582
Bars,
Application status, 22
Menu, 22
Toolbars, 22
base-uri,
as MapForce function (in xpath2 | accessors library), 307
boolean,
as MapForce function (in core | conversion functions), 235

C

Catalogs, 439
customize, 444
environment variables, 446
in DTD, 440
in XML Schema, 440
structure, 442
CDATA,
section, 116
ceiling,
as MapForce function (in core | math functions), 257
char-from-code,
as MapForce function (in core | string functions), 295
Code generation, 96, 103
Code point,
collation, 162
code-from-char,
as MapForce function (in core | string functions), 296
Collation,
locale collation, 162
sort component, 162
unicode code point, 162
Comments,
add to target files, 115
Complex type,
sorting, 162
Component,
sort data, 162
Component Icon Reference, 67
Components,
Add Duplicate Input After, 455
Add Duplicate Input Before, 455
Add/Remove/Edit Database Objects, 455
adding to the mapping, 69
Align Tree Left, 455
Align Tree Right, 455
aligning, 71
basics, 71
Change Root Element, 455
changing settings, 71

Components,
Constant, 452
Create Mapping to EDI X12 997, 455
Create Mapping to EDI X12 999, 455
Database, 452
Database Table Actions, 455
deleted items, 90
deleting, 92
EDI, 452
Edit FlexText Configuration, 455
Edit Schema Definition in XMLSpy, 455
Excel 2007+ File, 452
Exception, 452
Filter: Nodes/Rows, 452
icon reference, 67
IF-Else Condition, 452
Insert Input, 452
Insert Output, 452
Join, 452
JSON Schema/File, 452
menu commands, 455
overview, 67
Properties, 455
Protocol Buffers File, 452
Query Database, 455
Refresh, 455
Remove Duplicate, 455
searching, 71
settings, 71
Simple Input, 452
Simple Output, 452
Sort: Nodes/Rows, 452
SQL/NoSQL-WHERE/ORDER, 452
structural, 67, 105, 106
Text File, 452
transformation, 67
Value-Map, 452
Variable, 452
Web Service Function, 452
Write Content as CDATA Section, 455
XBRL Document, 452
XML, 107
XML and XML Schema, 107, 111, 113, 115, 116, 117, 120
XML Schema, 107
XML Schema/File, 452

concat,
as MapForce function (in core | string functions), 296

Connection type,
copy-all, 86
matching-children, 84
mixed, 82
source-driven, 82
standard, 81
standard with mixed content, 82
target-driven, 81
target-driven vs. source-driven, 82
target-driven with mixed content, 82

Connections,
annotation, 87
Auto Connect Matching Children, 457
change, 78
Connect Matching Children, 457
context menu, 89
copy, 78
copy-all, 81, 86
Copy-all (Copy Child Items), 457
create, 78
delete, 78
fix, 90
fixing after editing schema, 90
highlight selectively, 78
keeping after deleting components, 92
mandatory inputs, 78
matching-children, 81, 84
missing parent connections, 78
mixed, 81
move, 78
moving, 90
Properties, 457
see connection tooltips, 78
settings, 87
Settings for Connect Matching Children, 457
Source Driven (Mixed Content), 457
source-driven, 81
standard, 81
Target Driven (Standard), 457
target-driven, 81
types, 81, 87

Constants,
add, 191

contains,
as MapForce function (in core | string functions), 298

Conventions, 14

Copyright information, 584

count,
as MapForce function (in core | aggregate functions), 229

current,
as MapForce function (in xslt | xslt functions library), 366
current-date,
as MapForce function (in xpath2 | context functions), 312
current-dateTime,
as MapForce function (in xpath2 | context functions), 312
current-time,
as MapForce function (in xpath2 | context functions), 312
Customize,
commands, 464
context menus, 464
Default Menu vs. MapForce Design, 464
delete commands, 464
Keyboard, 465
menu shadows, 464
menus, 464
reset menu bars, 464
shortcuts, 465

D

Data streaming,
definition, 69
default-collation,
as MapForce function (in xpath2 | context functions), 312
Delete,
missing items, 90
Derived types,
map to/from, 111
xsi:type, 111
distinct-values,
as MapForce function (in core | sequence functions), 269
Distribution,
of Altova's software products, 584
divide,
as MapForce function (in core | math functions), 258
document,
as MapForce function (in xslt | xslt functions library), 366
DoTransform.bat,
execute with RaptorXML Server, 426
DTD,
source and target, 106
Duplicate input, 42

E

Edit,
Cut/Copy/Paste/Delete, 451
Find, 451
Find Next, 451
Find Previous, 451
Redo, 451
Select all, 451
Undo, 451
element-available,
as MapForce function (in xslt | xslt functions library), 367
End User License Agreement, 584, 586
equal,
as MapForce function (in core | logical functions), 251
equal-or-greater,
as MapForce function (in core | logical functions), 252
equal-or-less,
as MapForce function (in core | logical functions), 252
Errors,
out-of-memory, 69
troubleshooting, 69
Evaluation period,
of Altova's software products, 584
exists,
as MapForce function (in core | sequence functions), 271
Extension functions for XSLT and XQuery, 564
Extension Functions in .NET for XSLT and XQuery,
see under .NET extension functions, 573
Extension Functions in Java for XSLT and XQuery,
see under Java extension functions, 564
Extension Functions in MSXSL scripts, 579

F

false,
as MapForce function (in xpath2 | boolean functions), 310
Faulty connections,
after changing schema, 90
in databases, 90
in XML files, 90
File,
as a button in a component, 71

File,
as button on components, 417
Close, 448
Close All, 448
Compile to MapForce Server Execution File, 448
Deploy to FlowForce Server, 448
Exit, 448
Generate Code, 448
Generate Documentation, 448
Mapping Settings, 448
New, 448
Open, 448
Open Credentials Manager, 448
Print, 448
Print Preview, 448
Print Setup, 448
Recent files, 448
Reload, 448
Save, 448
Save All, 448
Save As, 448
Validate Mapping, 448
File names,
supplying as mapping input parameters, 421
File paths,
absolute, 73
broken, 73
fix broken references, 73
in execution environments, 76
in generated code, 76
of file-based databases, 73
relative, 73
relative versus absolute, 76
File/String,
as a button in a component, 71
as button on components, 417
File: (default),
as name of root node, 417
File: <dynamic>,
as name of root node, 417
Filtering,
data from components, 168
database tables, 168
Filters,
adding to the mapping, 168
first-items,
as MapForce function (in core | sequence functions), 273
floor,
as MapForce function (in core | math functions), 258
Folders,
as global resources, 437
format-date,
as MapForce function (in core | conversion functions), 235
format-dateTime,
as MapForce function (in core | conversion functions), 236
format-number,
as MapForce function (in core | conversion functions), 239
format-time,
as MapForce function (in core | conversion functions), 242
function-available,
as MapForce function (in xslt | xslt functions library), 367
Functions, 190
add, 191
add parameters, 191
argument data type, 191
basics, 191
constants, 191
Create User-Defined Function, 458
Create User-Defined Function from Selection, 458
delete parameters, 191
description, 191
find in the Libraries window, 191
find occurences in active mapping, 191
Function Settings, 458
Insert Input, 458
Insert Output, 458
parameters, 191
Remove Function, 458
search, 191

G

generate-id,
as MapForce function (in xslt | xslt functions library), 368
generate-sequence,
as MapForce function (in core | sequence functions), 274
get-fileext,
as MapForce function (in core | file path functions), 245
get-folder,
as MapForce function (in core | file path functions), 245
Global Resources,
creating, 430
Definitions file, 430
folders as, 437

Global Resources,
introduction to, 429
setup, 430
XML Files as, 435
greater,
as MapForce function (in core | logical functions), 253
group-adjacent,
as MapForce function (in core | sequence functions), 275
group-by,
as MapForce function (in core | sequence functions), 277
group-ending-with,
as MapForce function (in core | sequence functions), 279
group-into-blocks,
as MapForce function (in core | sequence functions), 280
group-starting-with,
as MapForce function (in core | sequence functions), 282
GUI, 21

H

Help,
About MapForce, 474
Check for Updates, 474
Download Components and Free Tools, 474
FAQ on the Web, 474
Index, 474
MapForce on the Internet, 474
MapForce Training, 474
Order Form, 474
Registration, 474
Search, 474
Software Activation, 474
Support Center, 474
Table of Contents, 474

I

If-Else conditions,
adding to the mapping, 168
implicit-timezone,
as MapForce function (in xpath2 | context functions), 313
Input,
duplicate, 71
Integration,
with Altova products, 19
Internet usage,
in Altova products, 583
is-xsi-nil,
as MapForce function (in core | node functions), 263
Item,
missing, 90
item-at,
as MapForce function (in core | sequence functions), 284
items-from-till,
as MapForce function (in core | sequence functions), 285

J

Java,
VM library location, 469
Java extension functions,
constructors, 569
datatype conversions, Java to Xpath/XQuery, 572
datatype conversions, XPath/XQuery to Java, 571
for XSLT and XQuery, 564
instance methods, instance fields, 571
overview, 564
static methods, static fields, 570
user-defined class files, 566
user-defined JAR files, 568

K

Key,
sort key, 162
Key-value pairs,
using on the mapping, 174

L

last,
as MapForce function (in xpath2 | context functions), 313
last-items,
as MapForce function (in core | sequence functions), 286
Legal information, 584
less,
as MapForce function (in core | logical functions), 253

License, 586
information about, 584
License metering,
in Altova products, 585
Licensing, 474
Locale collation, 162
local-name-from-QName,
as MapForce function (in lang | QName functions), 268
logical-and,
as MapForce function (in core | logical functions), 254
logical-not,
as MapForce function (in core | logical functions), 254
logical-or,
as MapForce function (in core | logical functions), 255
Look-up tables,
using on the mapping, 174

M

main-mfd-filepath,
as MapForce function (in core | file path functions), 246
MapForce,
overview, 14
Mapping,
basics, 65
components, 65
connections, 65
connectors, 65
creating, 65
fundamentals, 65
parts, 65
settings, 103
source-driven - mixed content, 82
terminology, 65
terms, 65
validating, 94
Mapping context, 399
Mapping input,
supplying custom file name as, 421
Supplying multiple files as, 417, 419, 421
Mapping output,
Generating multiple files as, 417, 421
Mapping scenarios, 17
max,
as MapForce function (in core | aggregate functions), 230
max-string,
as MapForce function (in core | aggregate functions), 230
Memory requirements, 582
Menu commands, 447
Component, 455
Connection, 457
Cutomize, 464, 465
Edit, 451
File, 448
Function, 458
Help, 474
Insert, 452
Output, 459
Tools, 463
Tools | Customize, 464
Tools | Keyboard, 465
Tools | Options, 467
Tools | Options | Java, 469
Tools | Options | Network Proxy, 470
View, 461
Windows, 473
Menu reference, 447
mfd-filepath,
as MapForce function (in core | file path functions), 246
Microsoft SharePoint Server,
adding files as components from, 69
min,
as MapForce function (in core | aggregate functions), 231
min-string,
as MapForce function (in core | aggregate functions), 232
Missing items, 90
Mixed,
content mapping, 82
source-driven mapping, 82
Mixed content,
mapping, 82
with standard connections, 82
with target-driven connections, 82
modulus,
as MapForce function (in core | math functions), 259
msxsl:script, 579
multiply,
as MapForce function (in core | math functions), 260

N

Namespace URI,

Namespace URI,
DTD, 106
Namespaces,
custom, 120
declare manually, 120
namespace-uri-form-QName,
as MapForce function (in lang | QName functions), 268
Network proxy,
automatic, 470
configuration, 470
manual, 470
settings, 470
system, 470
New Features, 11
Version 2019, 13
Version 2020, 12
Version 2021, 12
Version 2022, 11
Version 2023, 11
Node names,
mapping data from/to, 380
node-name,
as MapForce function (in core | node functions), 265
as MapForce function (in xpath2 | accessors library), 308
node-name function,
alternatives to using, 380
normalize-space,
as MapForce function (in core | string functions), 299
not-equal,
as MapForce function (in core | logical functions), 256
not-exists,
as MapForce function (in core | sequence functions), 287
NULL,
attribute, 113
values, 113
values in databases, 113
number,
as MapForce function (in core | conversion functions), 243

O

Ordering data,
sort component, 162
OS,
for Altova products, 582
Output,
Built-in Execution Engine, 459
C#, 459
C++, 459
Insert/Remove Bookmark, 459
Java, 459
Next Bookmark, 459
Pretty-Print XML Text, 459
previewing, 94
Previous Bookmark, 459
Regenerate Output, 459
Remove All Bookmarks, 459
Run SQL/NoSQL-Script, 459
Save All Output Files, 459
Save Output File, 459
saving, 94
Text View Settings, 459
Validate Output File, 459
validating, 94
XQuery, 459
XSLT 1.0, 459
XSLT 2.0, 459
XSLT 3.0, 459

P

Panes,
DB Query, 27
Mapping, 27
Output, 27
StyleVision output, 27
XQuery, 27
XSLT, 27
Parameters,
supplying to the mapping, 139, 143
Parent context,
example, 404
Parser,
built into Altova products, 582
Paths,
in generated code, 103
making absolute, 103
Platforms,
for Altova products, 582
position,
as MapForce function (in core | sequence functions), 288
Priority context, 408

Priority context, 408
example, 410
Processing Instructions,
add to target files, 115
Processing Instructions and Comments,
mapping, 82

Q

QName,
as MapForce function (in lang | QName functions), 267
Question mark,
missing items, 90

R

RaptorXML Server,
executing a transformation, 426
Regular expressions,
using in mappings, 221
remove-fileext,
as MapForce function (in core | file path functions), 246
remove-folder,
as MapForce function (in core | file path functions), 247
replace-fileext,
as MapForce function (in core | file path functions), 247
replicate-item,
as MapForce function (in core | sequence functions), 290
replicate-sequence,
as MapForce function (in core | sequence functions), 292
resolve-filepath,
as MapForce function (in core | file path functions), 248
resolve-uri,
as MapForce function (in xpath2 | anyURI functions), 309
round,
as MapForce function (in core | math functions), 260
round-half-to-even,
as MapForce function (in xpath2 | numeric functions), 336
round-precision,
as MapForce function (in core | math functions), 261

S

Schema,
generate, 106
industry standard, 106
pre-packaged, 106
Schema Manager,
CLI Help command, 131
CLI Info command, 132
CLI Initialize command, 132
CLI Install command, 133
CLI List command, 133
CLI overview, 131
CLI Reset command, 134
CLI Uninstall command, 135
CLI Update command, 136
CLI Upgrade command, 136
how to run, 125
installing a schema, 129
listing schemas by status in, 127
overview of, 122
patching a schema, 129
resetting, 130
status of schemas in, 127
uninstalling a achema, 130
upgrading a schema, 129
Scripts in XSLT/XQuery,
see under Extension functions, 564
Search,
items within mapping components, 71
Sequence, 398
set-empty,
as MapForce function (in core | sequence functions), 293
Settings,
for mappings, 103
for output file, 103
for web service operation, 103
set-xsi-nil,
as MapForce function (in core | node functions), 265
Simple type,
sorting, 162
skip-first-items,
as MapForce function (in core | sequence functions), 294
Smart component deletion, 92
Software product license, 586

Sort,
sort component, 162
Sort key,
sort component, 162
Sort order,
changing, 162
Source, 15
Source-driven,
mixed-content mapping, 82
SQLite,
changing database path to absolute in generated code, 76
starts-with,
as MapForce function (in core | string functions), 300
static-node-annotation,
as MapForce function (in core | node functions), 266
static-node-name,
as MapForce function (in core | node functions), 266
string,
as MapForce function (in core | conversion functions), 244
as MapForce function (in xpath2 | accessors library), 308
string-join,
as MapForce function (in core | aggregate functions), 233
string-length,
as MapForce function (in core | string functions), 300
Structural components,
XML, 106
XML and XML Schema, 106
XML Schema, 106
substitute-missing,
as MapForce function (in core | sequence functions), 295
substitute-missing-with-xsi-nil,
as MapForce function (in core | node functions), 267
substring,
as MapForce function (in core | string functions), 301
substring-after,
as MapForce function (in core | string functions), 301
substring-before,
as MapForce function (in core | string functions), 302
subtract,
as MapForce function (in core | math functions), 261
sum,
as MapForce function (in core | aggregate functions), 234
system-property,
as MapForce function (in xslt | xslt functions library), 368

T

Table data,
sorting, 162
Target, 15
Target component,
changing the processing order of, 413
Technical Information, 582
Text View,
bookmarks, 96
end-of-line markers, 96
folding margin, 96
indentation guides, 96
line numbers, 96
pretty-printing, 96
search, 100
source folding, 96
syntax coloring, 96
text highlighting, 96
whitespace markers, 96
word wrapping, 96
zooming, 96
tokenize,
as MapForce function (in core | string functions), 303
tokenize-by-length,
as MapForce function (in core | string functions), 303
tokenize-regexp,
as MapForce function (in core | string functions), 304
Tools,
Active Configuration, 463
Create Reversed Mapping, 463
Customize, 463
Global Resources, 463
menu command, 463
Options, 463
Restore Toolbars and Windows, 463
XBRL Taxonomy Manager, 463
Tools | Options,
Database, 467
Debugger, 467
Editing, 467
General, 467
Generation, 467
Java, 467
Messages, 467

Tools | Options,
Network Proxy, 467
XBRL, 467
Transformation languages,
BUILT-IN, 16
C#, 16
C++, 16
Java, 16
XQuery, 16
XSLT 1.0, 16
XSLT 2.0, 16
XSLT 3.0, 16
Transformations,
RaptorXML Server, 426
translate (in core | string functions),
as MapForce function, 306
true,
as MapForce function (in xpath2 | boolean functions), 310

U

UDFs,
and mapping context, 401
Unicode,
code point collation, 162
Unicode support,
in Altova products, 583
unparsed-entity-uri,
as MapForce function (in xslt | xslt functions library), 369
URI,
in DTDs, 106
URL,
adding files as components from, 69
User interface, 21
User-defined functions,
add parameters, 204
advantages, 198
call, 199
call recursively, 207
complex-type structures, 204
copy-paste, 199
create, 199
delete, 199
edit, 199
example, 198
examples, 207, 210
import, 199
inline, 199
input parameters, 199
lookup, 210
navigate, 199
of complex type, 204
of simple type, 204
output parameters, 199
overview, 198
parameter order, 204
parameters, 204
recursive, 207
recursive search, 207
regular, 199

V

Validate,
mapping design, 94
mapping output, 94
Validator,
in Altova products, 582
Value-Map,
as mapping component, 174
examples, 177, 180
Variables,
adding to the mapping, 152
changing the scope of, 156
complex, 150
DB-based, 150
examples of use, 158, 159
simple, 150
View,
Back, 461
Debug Windows, 461
Forward, 461
Libraries, 461
Manage Libraries, 461
menu command, 461
Messages, 461
Overview, 461
Project Window, 461
Show Annotations, 461
Show Connections from Source to Target, 461
Show Library in Function Header, 461
Show Selected Components Connectors, 461

View,
Show Tips, 461
Show Types, 461
Status Bar, 461
XBRL Display Options, 461
Zoom, 461

W

WebDAV Server,
adding files as components from, 69
Wildcards,
import schema, 117
selections, 117
wrapper schema, 117
xs:any/xs:anyAtrribute, 117
Windows,
Cascade, 473
Classic theme, 473
Dark theme, 473
Libraries, 22
Light theme, 473
Manage Libraries, 22
Mapping, 22
Messages, 26
Multiple Mapping Windows, 22
Overview, 22
Project, 22
support for Altova products, 582
Theme, 473
Tile Horisontal/Vertical, 473
Windows dialog, 473

X

XML,
component settings, 107
declaration, 107
digital signature, 107
encoding settings, 107
schema version, 103
XML Files,
as global resources, 435
generate from single XML source, 423
XML Parser,
about, 582
XML Schema Manager, 106
XQuery,
Extension functions, 564
xs: any, 117
xs:anyAttribute, 117
xsi:nil,
as attribute in XML instance, 113
XSLT,
adding custom functions, 214
Extension functions, 564
removing custom functions, 214
template namespace, 214

Z

Z to A,
sort component, 162