DB2XML User Guide

RenderX DB2XML User Guide

RenderX

Telephone: 1 (650) 328-8000
Fax: 1 (650) 328-8008
Website: http://renderx.com
Email: support@renderx.com

Table of Contents

Chapter 1. Overview

1.1. Introduction

This section contains introductory information about DB2XML.

DB2XML is a collection of libraries for generation of XML output from databases and another data sources.

The concept is to: 1.Design the structure of the output XML file using placeholders for data values, and; 2.Generate XML files by filling the placeholders using values from input data streams.

This is a common technique applicable to variable data publishing and other template-based conversions for data streams. In other words, it maps data to certain XML nodes and attributes.

Currently, DB2XML supports common SQL databases and CSV (Comma-Separated Values) files as data sources. The full list of supported databases, data source-specific parameters is defined in the Chapter 3, *Data Sources*. For XML generation, it supports XML attributes, elements, text and processing instruction nodes, conditional generation, value converters to handle specific tasks, and data grouping to make hierarchical trees from flat input data. The full list of features is described in the Chapter 4, *Data Mapping*.

DB2XML has a powerful and easy-to use graphical user interface to specify data sources, define mapping rules, and generate XML files. It is described in the Chapter 2, *DB2XML GUI Application*.

Basically, DB2XML jobs are defined as DB2XML tickets. DB2XML GUI Application uses ticket file format to store/load projects. The full description of ticket file format with examples is available in the Chapter 5, *Job Ticket File Format*.

1.2. Requirements

1.2.1. System Requirements

DB2XML runs on the Windows family of operating system including Windows XP, Windows Vista, Windows 7, and Windows 2008 Server. It requires .NET Framework Version 4.0 Full Profile to be installed on the target computer. To run DB2XML installer, you need Windows Installer 3.1 or later (usually it is a part of standard windows installation).

If you do not have .NET Framework Version 4.0 Full Profile installed, DB2XML installer automatically detects it and suggests to download and install it.

1.2.2. Optional Components

To connect to an SQL database, the appropriate Data Provider should be registered in the system. Data Providers that require installation are:

- To use Oracle as Data Source, one of the following Data Providers should be installed:
  - Oracle Data Provider for .NET / ODP.NET (for Oracle9i Release 2 or later), which can be downloaded from Oracle's website .
  - .NET Framework Data Provider for Oracle (version 8i Release 3 (8.1.7) or later), which is included in .NET Framework v2.0 or later and does not require any installation. For .NET Framework v1.0, .Net Framework Data Provider for Oracle should be installed separately. It can be downloaded from Microsoft.
- To use MySQL as Data Source, MySql Connector/Net should be installed which can be downloaded from MySQL website.

Note: Both Oracle Data Providers require Oracle Client to be installed. For more information on Oracle Data Access Components, visit Oracle Data Provider for .NET

1.3. Installation

DB2XML is distributed as a singe ZIP archive file. For proper running, the application requires a license file, which can be obtained from RenderX.

1.3.1. Common Steps

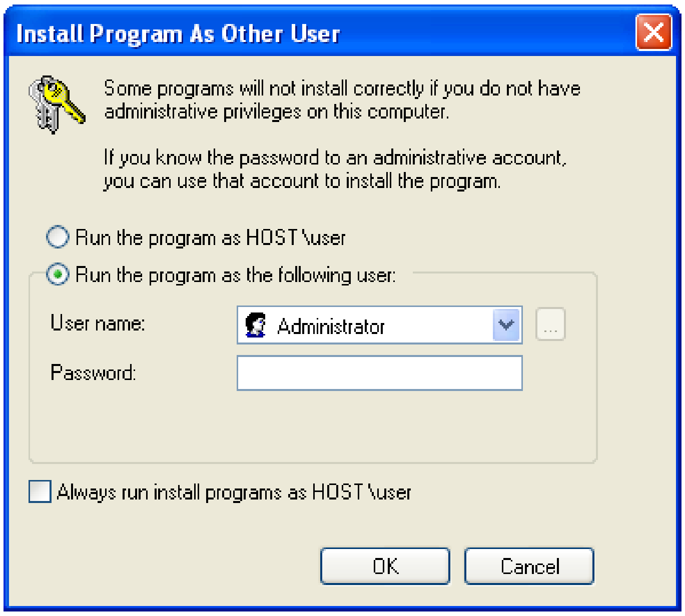
To install DB2XML on your Windows operating system:

1. Extract the `RenderXDB2XML-X.X.X.X.zip` file.
2. In Windows Explorer or another file manager, go to the directory which contains the extracted files and run `Setup.exe` either by double-clicking it or by selecting it and pressing the Enter key. It will check for prerequisites for DB2XML and suggest to install the applicable components, if necessary, and then run DB2XML installer.
3. Follow the wizard to install prerequisites and then DB2XML on your computer.
4. Copy license.xml file obtained from RenderX to the installation folder (`%ProgramFiles%\RenderX\DB2XML by default`).

1.3.2. Installing on Windows XP

DB2XML requires an administrator account to install it on Windows XP. In step 2, when `Setup.exe` is run from a non-administrator account, it will open the "Install Program As Other

User" dialog. Choose an administrator user from the drop down list and specify password for the selected user.



Some programs will not install correctly if you do not have administrative privileges on this computer.

If you know the password to an administrative account, you can use that account to install the program.

○ Run the program as HOST\user
◉ Run the program as the following user:

User name: Administrator
Password:

☐ Always run install programs as HOST\user

Figure 1.1. "Install As" dialog.

1.3.3. Installing on Windows Vista and Windows 7

To install DB2XML to a Windows Vista or Windows 7 machine from a user account, you need to have UAC (User Account Control) to be turned on. DB2XML installer opens "Install As" dialog, where you need to choose an administrator user and password. When installed from an administrator account, the DB2XML installer may request elevation depending on UAC settings.

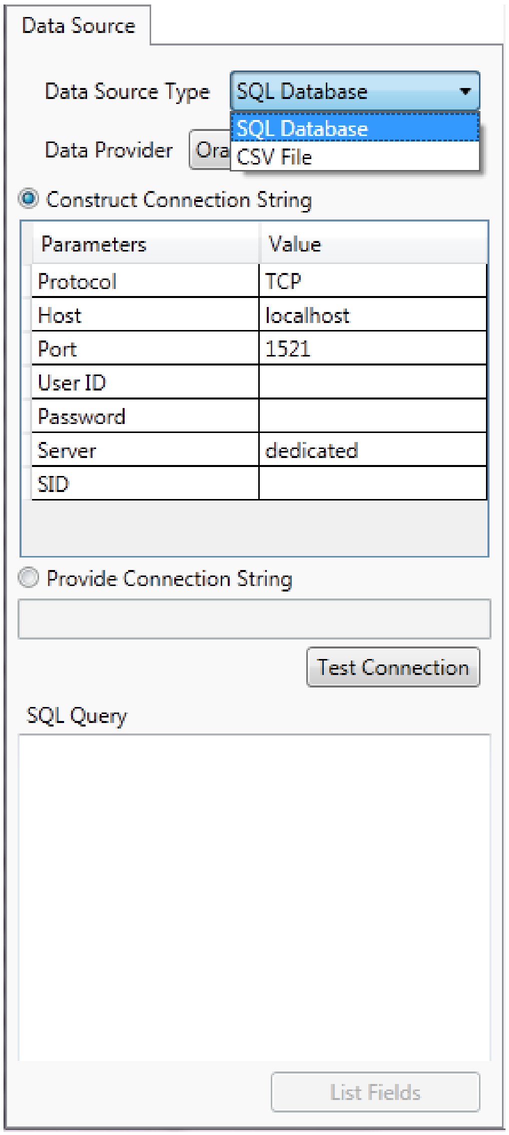
1.3.4. Uninstalling DB2XML

To uninstall DB2XML, go to Start -> Control Panel -> Programs and Features (Add or Remove Programs in Windows XP). RenderX DB2XML is located in the list of programs. Right-click it and then click "Uninstall". For Windows XP, an administrator account is needed to uninstall DB2XML. For Windows Vista/7, the elevation dialog is prompted when uninstalling DB2XML from a user account.

Chapter 2. DB2XML GUI Application

2.1. Data Sources

The Data Source panel contains data source connection parameters.



Data Source Type: SQL Database

Data Provider: Ora

● Construct Connection String

Parameters	Value
Protocol	TCP
Host	localhost
Port	1521
User ID	
Password	
Server	dedicated
SID	

○ Provide Connection String

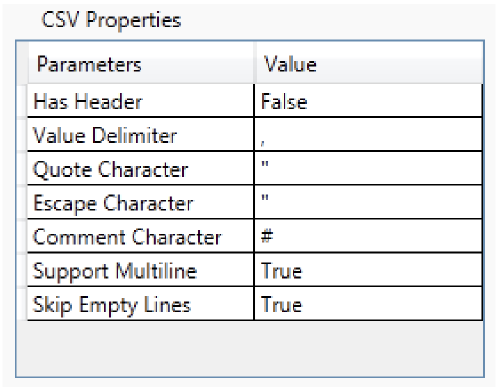
Figure 2.1. Data Source Section

The first step of connection definition is the data source selection. DB2XML supports SQL Databases and CSV files as data sources. If the SQL Database data source type is selected, it is necessary to select the data provider. If the CSV file data source type is selected, the CSV file should be selected for further processing.

After data source selection, a specific database connection options table appears.

For detailed information on supported data sources and options descriptions, see the Chapter 3, *Data Sources*.

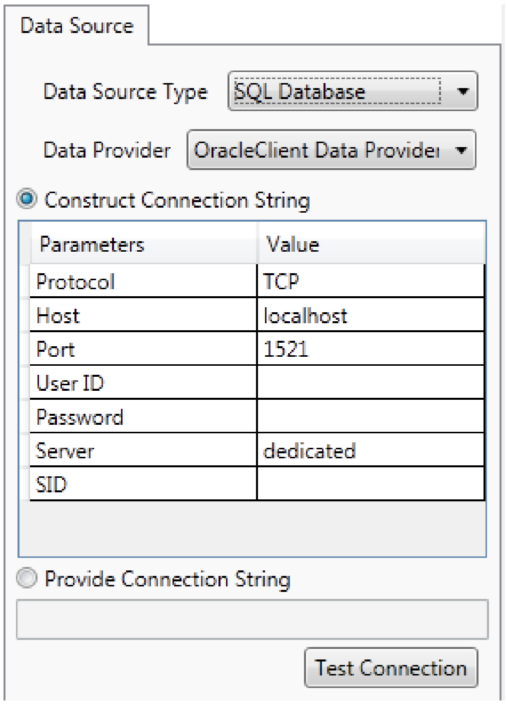
2.2. Data Source Parameters



CSV Properties

Parameters	Value
Has Header	False
Value Delimiter	,
Quote Character	"
Escape Character	"
Comment Character	#
Support Multiline	True
Skip Empty Lines	True

Figure 2.2. CSV File Parameters



Data Source

Data Source Type: SQL Database

Data Provider: OracleClient Data Provider

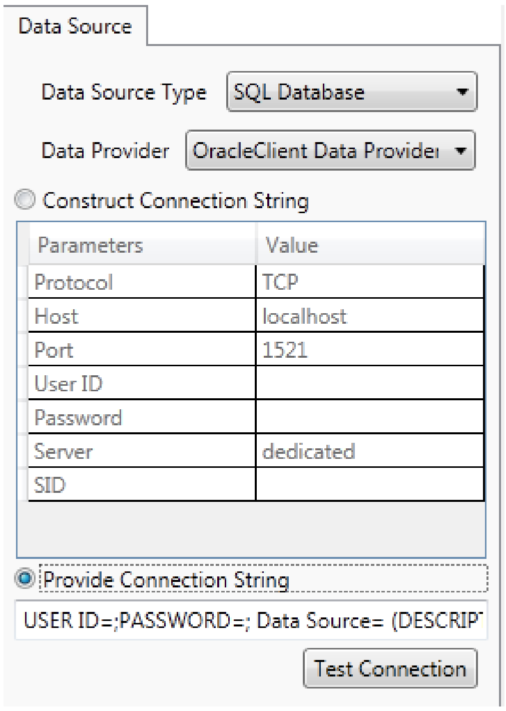
(●) Construct Connection String

Parameters	Value
Protocol	TCP
Host	localhost
Port	1521
User ID	
Password	
Server	dedicated
SID	

( ) Provide Connection String

Figure 2.3. SQL Database Parameters

For SQL databases, it is also possible to provide a complete connection string instead of setting up parameters manually. To use the connection string, check the Provide Connection String radio button.



Data Source Type: SQL Database

Data Provider: OracleClient Data Provider

Construct Connection String

Parameters	Value
Protocol	TCP
Host	localhost
Port	1521
User ID	
Password	
Server	dedicated
SID	

[✓ Provide Connection String]

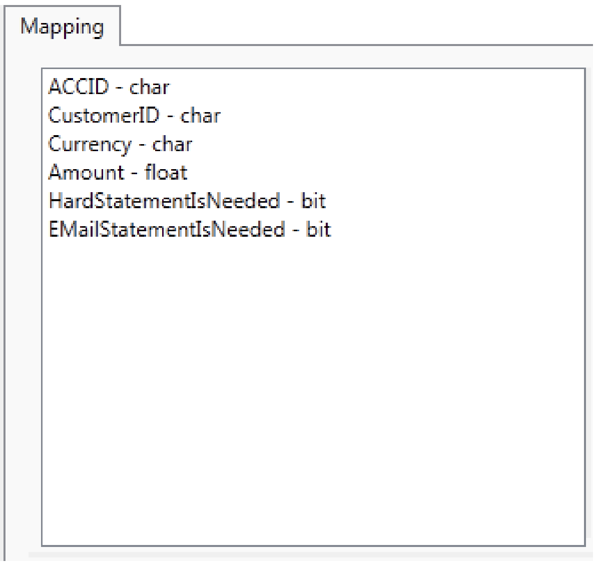
USER ID=;PASSWORD=; Data Source= (DESCRIP

Figure 2.4. Providing Connection String for SQL databases

After successful connection to the specified database, to get database fields for further mapping definition, the SQL Query should be defined and fields should be listed.

2.3. Data Fields

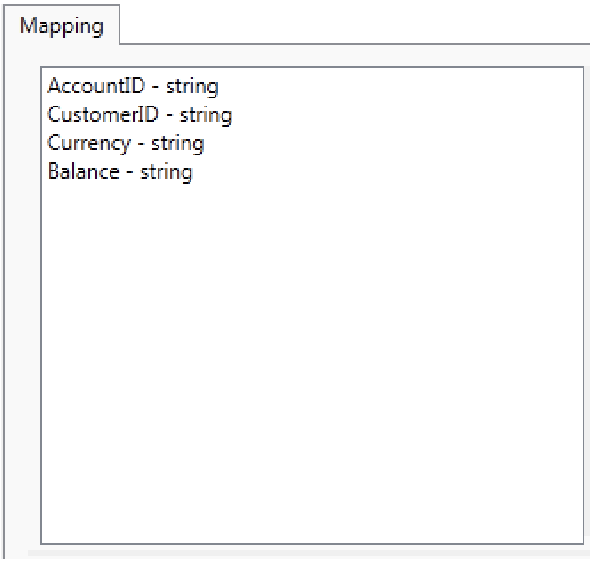
After execution of the "List Fields" command, the data fields appear in Mapping tab.



ACCID - char
CustomerID - char
Currency - char
Amount - float
HardStatementIsNeeded - bit
EMailStatementIsNeeded - bit

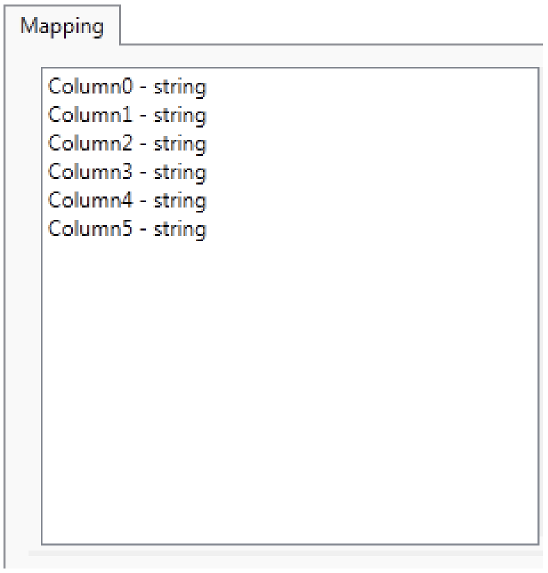
Figure 2.5. Fields list

For the CSV files, if the `Has Header` option set to True, and the CSV file contains headers, the fields list shows the column headers from the CSV file; in the opposite case, the fields list shows *`ColumnNumber`* for each column.



AccountID - string
CustomerID - string
Currency - string
Balance - string

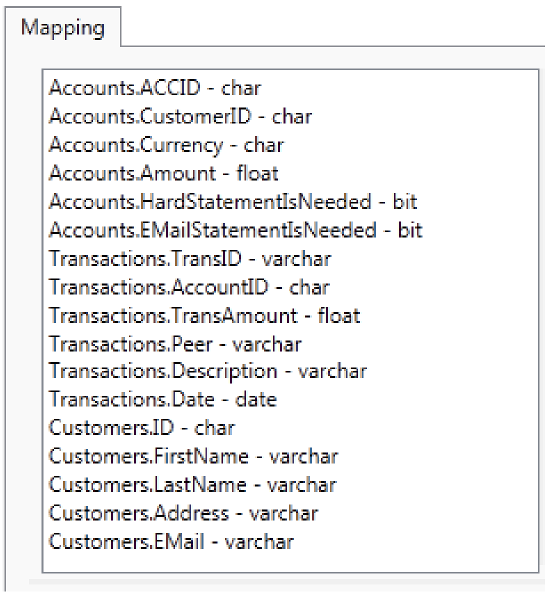
Figure 2.6. CSV file fields list if `Has header=True`



Column0 - string
Column1 - string
Column2 - string
Column3 - string
Column4 - string
Column5 - string

Figure 2.7. CSV file fields list for `Has Header=False`

For SQL database fields, the list shows only table field names (as shown in the "Fields List" figure above), if the SQL query selects only one table from the database, and *`Tablename.Fieldname`*, if the SQL query selects more than one table from the database.



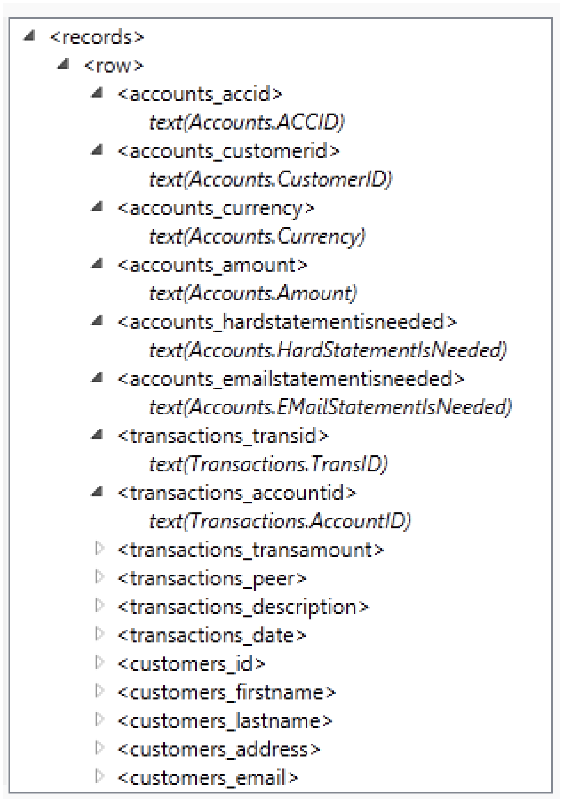
Accounts.ACCID - char
Accounts.CustomerID - char
Accounts.Currency - char
Accounts.Amount - float
Accounts.HardStatementIsNeeded - bit
Accounts.EMailStatementIsNeeded - bit
Transactions.TransID - varchar
Transactions.AccountID - char
Transactions.TransAmount - float
Transactions.Peer - varchar
Transactions.Description - varchar
Transactions.Date - date
Customers.ID - char
Customers.FirstName - varchar
Customers.LastName - varchar
Customers.Address - varchar
Customers.EMail - varchar

Figure 2.8. Fields list for multiple tables

The fields are draggable and can be moved to the Mapping Rules tree to be added as mapping rule items.

2.4. Mapping Rules

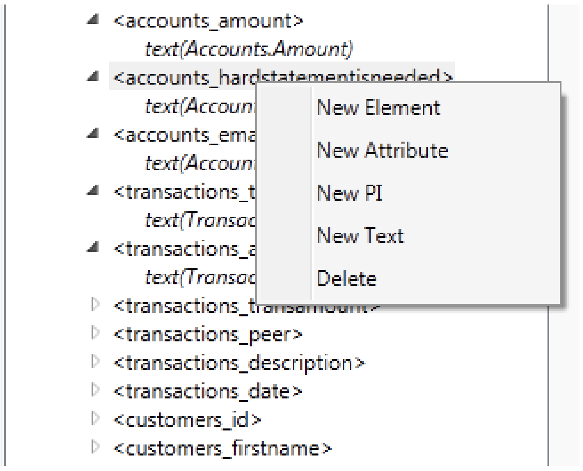
Mapping is a set of rules to define structure of the final XML file to be generated. For the detailed information see the Chapter 4, *Data Mapping*.



```
◢ <records>
  ◢ <row>
    ◢ <accounts_accid>
        text(Accounts.ACCID)
    ◢ <accounts_customerid>
        text(Accounts.CustomerID)
    ◢ <accounts_currency>
        text(Accounts.Currency)
    ◢ <accounts_amount>
        text(Accounts.Amount)
    ◢ <accounts_hardstatementisneeded>
        text(Accounts.HardStatementIsNeeded)
    ◢ <accounts_emailstatementisneeded>
        text(Accounts.EMailStatementIsNeeded)
    ◢ <transactions_transid>
        text(Transactions.TransID)
    ◢ <transactions_accountid>
        text(Transactions.AccountID)
    ▷ <transactions_transamount>
    ▷ <transactions_peer>
    ▷ <transactions_description>
    ▷ <transactions_date>
    ▷ <customers_id>
    ▷ <customers_firstname>
    ▷ <customers_lastname>
    ▷ <customers_address>
    ▷ <customers_email>
```

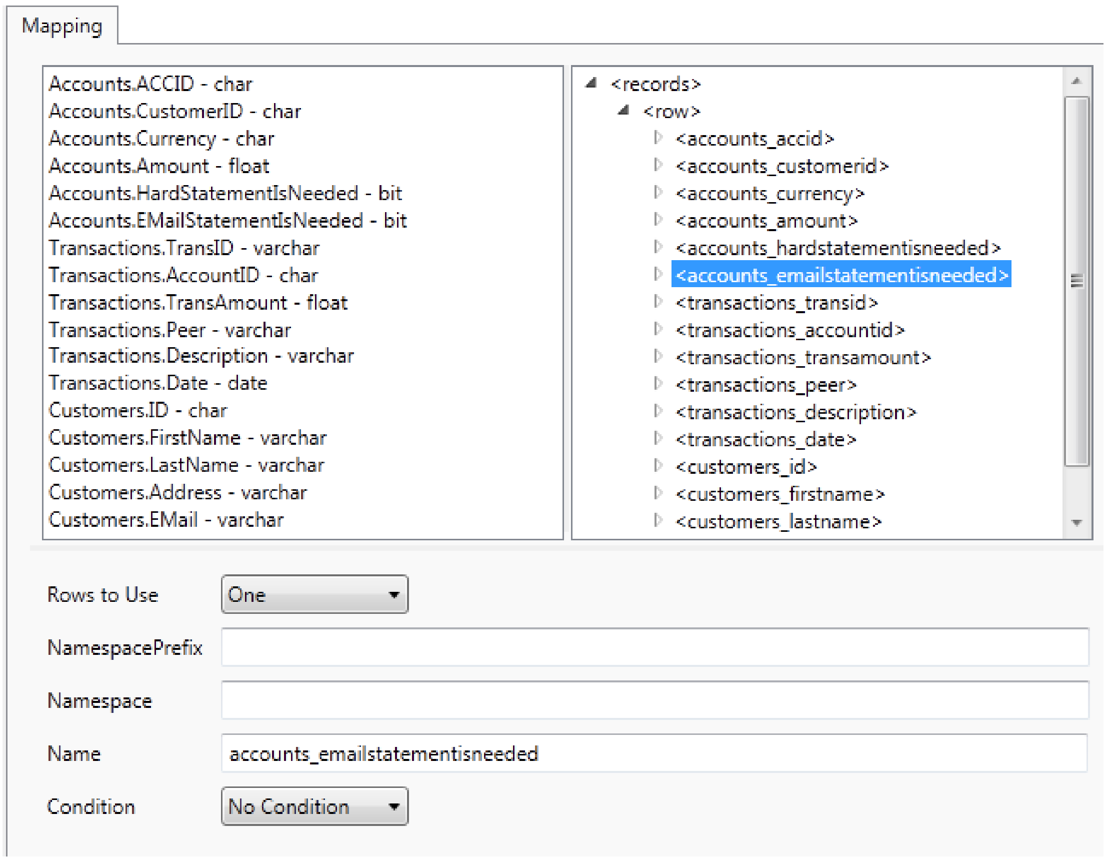
Figure 2.9. Flat Mapping

Default mapping is a flat mapping with elements containing field values as text items. It is possible to modify the mapping rules by using the context menu, by draging and droping items, and changing the properties of selected items.



```
◢ <accounts_amount>
    text(Accounts.Amount)
◢ <accounts_hardstatementisneeded>
    text(Accoun
◢ <accounts_ema
    text(Accoun
◢ <transactions_t
    text(Transac
◢ <transactions_a
    text(Transac
▷ <transactions_transamount>
▷ <transactions_peer>
▷ <transactions_description>
▷ <transactions_date>
▷ <customers_id>
▷ <customers_firstname>
```

Figure 2.10. Mapping Context Menu



Accounts.ACCID - char
Accounts.CustomerID - char
Accounts.Currency - char
Accounts.Amount - float
Accounts.HardStatementIsNeeded - bit
Accounts.EMailStatementIsNeeded - bit
Transactions.TransID - varchar
Transactions.AccountID - char
Transactions.TransAmount - float
Transactions.Peer - varchar
Transactions.Description - varchar
Transactions.Date - date
Customers.ID - char
Customers.FirstName - varchar
Customers.LastName - varchar
Customers.Address - varchar
Customers.EMail - varchar

```
<records>
  <row>
    <accounts_accid>
    <accounts_customerid>
    <accounts_currency>
    <accounts_amount>
    <accounts_hardstatementisneeded>
    <accounts_emailstatementisneeded>
    <transactions_transid>
    <transactions_accountid>
    <transactions_transamount>
    <transactions_peer>
    <transactions_description>
    <transactions_date>
    <customers_id>
    <customers_firstname>
    <customers_lastname>
```

Rows to Use	One
NamespacePrefix	
Namespace	
Name	accounts_emailstatementisneeded
Condition	No Condition

Figure 2.11. Mapping Item Properties

2.5. Number of Records

It is possible to limit number of generated records (top-level items). Number of records may be `all` or any valid positive integer.

2.6. Output File

The output file specifies the path to the XML output file to be generated.

2.7. Designing Stylesheet in VisualXSL

To design stylesheet for the generated file, DB2XML generates a simple XML file with the defined hierarchy, creates a simple VXSL Project and opens application based on that project. The created VXSL project is based on the generated simple XML file and contains structure of XML file.

2.8. Load/Save Project Files

DB2XML GUI Application stores and loads projects using DB2XML Job Ticket format. The default extension for this format is .dxt.

Chapter 3. Data Sources

In DB2XML, Data Source is used to identify data warehouses, physical data repositories such as complex databases, or simple files with rows and columns. Every selected Data Source should have its own connection properties specified. DB2XML uses appropriate data providers to connect to a Data Source and retrieve data. There are two types of Data Sources supported by DB2XML:

- SQL database
- CSV file

3.1. SQL Database

The SQL database Data Source is used to retrieve data from SQL Database Management Systems, such as Oracle and Microsoft SQL Server. DB2XML uses the ADO.NET database access technology to interact with Database Management Systems. DB2XML uses Data Providers installed on the system to retrieve data from Data Sources. The currently supported Data Providers are:

- Oracle Data Provider for .NET / ODP.NET
- .NET Framework Data Provider for Oracle
- SQL Client Data Provider
- MySQL Data Provider

Each Data Provider has a set of connection string parameters that need to be provided. The parameters can be set or connection string can be specified directly. Also SQL Query should be provided to retrieve appropriate fields from the Data Source after connection has been established.

3.1.1. Oracle Data Provider for .NET (ODP.NET)

The Oracle Data Provider for .NET (ODP.NET) features optimized data access to the Oracle database from a .NET environment developed and supported by Oracle.

The ODP.NET Data Provider makes access to Oracle databases from .NET more flexible, faster, and more stable. ODP.NET is designed for scalable enterprise Windows solutions by providing full support for Unicode and local and distributed transactions. For more information, visit Oracle Data Provider for .NET.

Connection string parameters for ODP.NET are:

Table 3.1. ODP.NET Parameters

Parameter	Description	Values
Protocol	Network protocol used to connect to the server.	*TCP*, *UDP*. Default value is *TCP*.
Host	Host to connect to.	Default value is *localhost*.
Port	Port Oracle server is running on.	Default value is *1521*.
User ID	User's credentials.	For example *SYSDBA*.
Password	The password for the Oracle account logging on.	
Server	Specifies whether the connection is established using a shared or dedicated server process.	*dedicated*, *shared*. Default value is *dedicated*.
SID	Oracle System Identifier that refers to the instance of the Oracle database software running on the server.	Default value is *orcl*.

3.1.2. .NET Framework Data Provider for Oracle

The Microsoft .NET Framework Data Provider for Oracle is a component included in the .NET Framework that provides access to an Oracle database using the Oracle Call Interface (OCI) as provided by Oracle Client software. The .NET Framework Data Provider for Oracle supports Oracle client software version 8i Release 3 (8.1.7) and later.

Connection String Parameters for .NET Framework Data Provider for Oracle are the same as for ODP.NET

3.1.3. SQL Client Data Provider

The Microsoft SQL Server .NET Data Provider allows you to connect to Microsoft SQL Server (v7.0 and later) databases.

Connection string parameters used to connect to a SQL Server are:

Table 3.2. SQL Client Data Provider Parameters

Parameter	Description	Values
Data Source	The name or network address of the instance of SQL Server to which to connect.	Default value is *.\SQLEXPRESS*

Parameter	Description	Values
Port	Port SQL Server instance is running on.	Default value is *1433*
User ID	The SQL Server login account.	
Password	The password for the SQL Server account logging on.	
Initial Catalog	The name of the database.	
Integrated Security	When *False*, *User ID* and *Password* are specified in the connection. When *True* , the current Windows account credentials are used for authentication. If *User ID* and *Password* are specified and *Integrated Security* is set to *True* , the *User ID* and *Password* will be ignored and *Integrated Security* will be used.	*True*, *False*, *Yes*, *No* and *sspi* (equivalent to true). Default value is *True*.
Connect Timeout	The length of time (in seconds) to wait for a connection to the server before terminating the attempt and generating an error.	Default value is *30*.
User Instance	A value that indicates whether to redirect the connection from the default SQL Server Express instance to a runtime-initiated instance running under the account of the caller.	Default value is *True*.

3.1.4. MySQL Data Provider

This provider is required to use MySQL database as Data Source. Visit MySQL Connector/Net for more information.

Connection string parameters used to connect to a MySQL instance are:

Table 3.3. MySQL Data Provider Parameters

Parameter	Description	Values
Data Source	The name or network address of the instance of MySQL to which to connect.	Default value is *localhost*.
Port	Port MySQL instance is running on.	Default value is *3306.*
User ID	The MySQL login account.	

Parameter	Description	Values
Password	The password for the MySQL account logging on.	
Initial Catalog	The name of the database.	
Connect Timeout	The length of time (in seconds) to wait for a connection to the server before terminating the attempt and generating an error.	Default value is *15*.

3.2. CSV Files

CSV (stands for Comma-Separated Values) is a file format which is used to store information in a very simple way. It is often used to move tabular data between different computer programs. CSV has low overhead, thereby allows transfer data with less bandwidth.

DB2XML uses CSV Reader as CSV file data reader which performs fast data extraction with low memory cost. The required parameter which should be specified is CSV file to read data from.

Depending on the CSV file format (e.g. value delimiter, quote character, etc.), the following configurable options can be set:

Table 3.4. CSV Reader Parameters

Parameter	Description	Default Value
Has Header	Indicates if field names are located on the first non commented line.	*False*
Value Delimiter	Delimiter character separating every field.	*,*
Quote Character	Quote character wrapping every field.	*"*
Escape Character	Escape character letting insert quotation characters inside a quoted field.	*"*
Comment Character	Comment character indicating that a line is commented out.	*#*
Support Multiline	Turns the multiline support on/off.	*True*
Skip Empty Lines	Defines skip empty lines or not.	*True*

Chapter 4. Data Mapping

Mapping rules define the ways to generate XML nodes and attributes from the input data stream. The result XML document is composed by applying the mapping rules to the data stream. In this topic we will use DB2XML job ticket format to define mapping rules. It is also possible to declare mapping rules using DB2XML API.

4.1. Data Fields

Simply, Data Fields are columns from input data stream.

For SQL database data sources, a data field contains information about the fully qualified column name and its data type.

For a CSV file data source, all columns are treated as strings. If a CSV file contains column headers in the first row, they can be used for column names. When a CSV file does not contain column names, DB2XML generates field names as `Column0`, `Column1`, ... `ColumN`, where indices start from 0.

4.1.1. Examples

SQL Database Sample

Consider a sample SQL database (SAMPLE) with the following tables:

Table 4.1. ***Employees***

ID (INTEGER)	Name (VARCHAR 255)	Role (VARCHAR 255)
1	"Joe Somebody"	"Engineer"
...	...	...

Table 4.2. ***Projects***

Name (VARCHAR 255)	Maintainer (INTEGER)
"DB2XML"	1
...	...

The following SQL query:

```
SELECT * FROM Employees
```

will return the following fields:

- *`ID`*

- *Name*
- *Role*

The following SQL *JOIN* query:

```
SELECT * FROM Projects JOIN Employees ON Projects.Maintainer=Employees.ID
```

will return the following fields:

- *Projects.Name*
- *Projects.Maintainer*
- *Employees.ID*
- *Employees.Name*
- *Employees.Role*

CSV File

With the following CSV file:

```
col1, col2, col3
a, 1, 2.4
b, 2, 3.5
c, 3, 4.6
```

we need to set Has Header parameter to *True* in the data source parameters to use *col1*, *col2*, *col3* as data field names; otherwise, the first row will be treated as data.

4.1.2. Data Field Types

All the data fields can be of certain data types. The Data Field type defines the way its values is treated in expressions.

Depending on the source data type, a field can be:

- *primitive* - basic primitive types: numerics, booleans, short strings.
- *complex*, consists of:
  - *binary* - byte arrays.
  - *long character data* - e.g. long strings.

The types are defined according to the following rules:

- All CSV data fields are of *string* type. *primitive*, not *binary*.

- Database data field is:
  - *`complex:`*
    - *`binary`*, if the corresponding database type is one of the following: *`varbinary`*, *`blob`*, *`image`*, *`binary`*, *`raw`*, *`long raw`*, *`bfile`*, *`longblob`*, *`long varbinary`*, *`mediumblob`*, *`tinyblob`*.
    - *`long character data`*, if the corresponding database type is one of the following: *`clob`*, *`nclob`*.
  - *`primitive`*, if it is not *`complex`*.

4.2. XML Structure

DB2XML supports generation of XML attributes and element, text and processing instruction nodes.

4.3. What is a Mapping Rule?

Mapping rule is a declaration of a particular item in the result XML document.

The following mapping rule:

```
<element name="firstname"/>
```

defines an XML element named firstname, from the default XML namespace, with no attributes or child nodes. In the result XML document it will appear as:

```
<firstname/>
```

The following mapping rule:

```
<element name="firstname">
  <text value="'John'"/>
  </element>
```

defines the same element with a single text child which has a constant value John. The result XML will be as follows:

```
<firstname>John</firstname>
```

These mapping rules:

```
<document encoding="UTF-8">
  <element name="people" rows-to-use="all">
    <element name="person">
      <element name="firstname">
        <text value="'John'"/>
      </element>
      <element name="lastname">
        <text value="'Smith'"/>
      </element>
    </element>
  </element>
</document>
```

will generate the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<people>
  <person>
    <firstname>John</firstname>
    <lastname>Smith</lastname>
  </person>
  ...
</people>
```

Where the `person` element and its content is repeated for each row of the input data stream.

The following example uses data fields from the *`Employees`* table from the SQL Database Sample above:

```
<document encoding="UTF-8">
  <element name="company">
    <processing-instruction target="role" data="Role"/>
    <element name="employee">
      <attribute name="employee-id" value="ID"/>
      <element name="name">
        <text value="Name"/>
      </element>
    </element>
  </element>
</document>
```

It will generate the following XML file:

```
<company>
  <?role Engineer?>
  <employee employee-id="1">
    <name>Joe Somebody</name>
  </employee>
  ...
</company>
```

Each `<?role>` processing instruction and `employee` element is repeated for each row in the original table: `<?role>` contains the appropriate *`Role`* value and `name` contains the employee's name from that table.

4.4. Populating XML Nodes

Consider the following sample data table (either in a database or CSV file):

Table 4.3. Sample Table

Column1	Column2	Column3
"a"	1	2.4
"a"	2	3.5
"a"	2	3.3
"b"	3	4.6
"b"	4	5.7
"c"	5	6.8

We will refer to this table in the examples below.

4.4.1. Generating "Flat" XML file

Now we want to generate a very simple XML file in which:

- Each row from the table goes as `row` XML element, and;
- Each field value goes as <*column_name*>*value*</*column_name*>,where:
    - `column_name` is the column name, and;
    - *value* is the value for the given column in the current row.

All the rows are collected into the `records` element.

The following ticket file snippet contains mapping rules:

```
<element name="records" rows-to-use="all">
  <element name="row" rows-to-use="one">
    <element name="column1" rows-to-use="one">
      <text converter="Auto" value="Column1"/>
    </element>
    <element name="column2" rows-to-use="one">
      <text converter="Auto" value="Column2"/>
    </element>
    <element name="column3" rows-to-use="one">
      <text converter="Auto" value="Column3"/>
    </element>
  </element>
</element>
```

Note that the `rows-to-use` attribute for an element define how the input data rows are consumed by that element.

Using the mapping rules above and the Table 4.3, "Sample Table", DB2XML will produce the following output XML file:

```
<records>
  <row>
    <column1>a</column1>
    <column2>1</column2>
    <column3>2.4</column3>
  </row>
  <row>
    <column1>a</column1>
    <column2>2</column2>
    <column3>3.5</column3>
  </row>
  <row>
    <column1>a</column1>
    <column2>2</column2>
    <column3>3.3</column3>
  </row>
  <row>
    <column1>b</column1>
    <column2>3</column2>
    <column3>4.6</column3>
  </row>
  <row>
    <column1>b</column1>
    <column2>4</column2>
    <column3>5.7</column3>
  </row>
  <row>
    <column1>c</column1>
    <column2>5</column2>
    <column3>6.8</column3>
  </row>
</records>
```

In this example, `row` element repeated 5 times, because there are 5 rows in the original table and each `row` element consumes one row from the input data stream.

In this example, `row` is the XML element which is being populated in the main loop. The main loop for XML content population happens inside the `records` element.

Here the `row` element acts as a *`wrapper`*, which means that it wraps the sequences of values and consumes one row.

4.4.2. Generating XML Using Grouping

Now consider the input data from the Table 4.3, “Sample Table” and imagine that we want to collect all the rows which contain the same value for the *`Column1`* field into one element. The appropriate mapping rules should be as follows:

```
<element name="records" rows-to-use="all">
  <element name="group" rows-to-use="unchanged(Column1)">
    <element name="column1" rows-to-use="one">
      <text converter="Auto" value="Column1"/>
    </element>
    <element name="column2" rows-to-use="one">
      <text converter="Auto" value="Column2"/>
    </element>
    <element name="column3" rows-to-use="one">
      <text converter="Auto" value="Column3"/>
    </element>
  </element>
</element>
```

With these mapping rules, each `group` element will consume all the rows from input data until the value of *`Column1`* is changed. In other words, these mapping rules generate distinct group elements for distinct values of *`Column1`*.

The result XML will be the following:

```
<records>
  <group>
    <column1>a</column1>
    <column2>1</column2>
    <column3>2.4</column3>
    <column1>a</column1>
    <column2>2</column2>
    <column3>3.5</column3>
    <column1>a</column1>
    <column2>2</column2>
    <column3>3.3</column3>
  </group>
  <group>
    <column1>b</column1>
    <column2>3</column2>
    <column3>4.6</column3>
    <column1>b</column1>
    <column2>4</column2>
    <column3>5.7</column3>
  </group>
  <group>
    <column1>c</column1>
    <column2>5</column2>
    <column3>6.8</column3>
  </group>
</records>
```

Here, there are 3 `group` elements for 3 different values of *`Column1`*: *`"a", "b", "c"`*. The first `group` element consumed three rows, the second one consumed two rows and the last one consumed one, according to the values in the input data stream.

4.4.3. Introducing *`Sequence Wrappers`*

`Sequence wrapper` is an element which consumes all the rows available in the parent context. To wrap sequences, the `rows-to-use` attribute should be set to *`all`* for the given element.

The following example shows how sequence wrappers work:

```
<element name="records" rows-to-use="all">
  <element name="group" rows-to-use="unchanged(Column1)">
    <element name="column1" rows-to-use="one">
      <text converter="Auto" value="Column1"/>
    </element>
    <element name="values" rows-to-use="all">
      <element name="column2" rows-to-use="one">
        <text converter="Auto" value="Column2"/>
      </element>
      <element name="column3" rows-to-use="one">
        <text converter="Auto" value="Column3"/>
      </element>
    </element>
  </element>
</element>
```

Note the *`<element name="values" rows-to-use="all">`* rule inside the `group` rule.

The result XML is the following:

```
<records>
  <group>
    <column1>a</column1>
    <values>
      <column2>1</column2>
      <column3>2.4</column3>
      <column2>2</column2>
      <column3>3.5</column3>
      <column2>2</column2>
      <column3>3.3</column3>
    </values>
  </group>
  <group>
    <column1>b</column1>
    <values>
      <column2>3</column2>
      <column3>4.6</column3>
      <column2>4</column2>
      <column3>5.7</column3>
    </values>
  </group>
  <group>
    <column1>c</column1>
    <values>
      <column2>5</column2>
      <column3>6.8</column3>
    </values>
  </group>
</records>
```

Here the values element is a *`sequence wrapper`*, which means that it populates its content for each data row in the parent context. Inside the `group` elements, the child population loop has exactly one iteration, because the `values` element consumes all the rows available in the parent context (the `group` element in this case).

4.4.4. Using Sequence Wrappers at the Root Level

Consider the following mapping rules:

```
<element name="records" rows-to-use="all">
  <element name="sequence-wrapper-in-root" rows-to-use="all">
    <element name="sequence-wrapper-in-sequence-wrapper" rows-to-use="all">
      <element name="row" rows-to-use="one">
        <element name="column1" rows-to-use="one">
          <text converter="Auto" value="Column1"/>
        </element>
        <element name="column2" rows-to-use="one">
          <text converter="Auto" value="Column2"/>
        </element>
        <element name="column3" rows-to-use="one">
          <text converter="Auto" value="Column3"/>
        </element>
      </element>
    </element>
  </element>
</element>
```

In the above example, there are two cascaded sequence wrappers:

- *sequence-wrapper-in-root*, and;
- *sequence-wrapper-in-sequence-wrapper*.

The result XML file generated using these rules will be the following:

```
<records>
  <sequence-wrapper-in-root>
    <sequence-wrapper-in-sequence-wrapper>
      <row>
        <column1>a</column1>
        <column2>1</column2>
        <column3>2.4</column3>
      </row>
      <row>
        <column1>a</column1>
        <column2>2</column2>
        <column3>3.5</column3>
      </row>
      <row>
        <column1>a</column1>
        <column2>2</column2>
        <column3>3.3</column3>
      </row>
      <row>
        <column1>b</column1>
        <column2>3</column2>
        <column3>4.6</column3>
      </row>
      <row>
        <column1>b</column1>
        <column2>4</column2>
        <column3>5.7</column3>
      </row>
      <row>
        <column1>c</column1>
        <column2>5</column2>
        <column3>6.8</column3>
      </row>
    </sequence-wrapper-in-sequence-wrapper>
  </sequence-wrapper-in-root>
</records>
```

Here the main loop happens inside the `sequence-wrapper-in-sequence-wrapper` element, because the maximum number of rows consumed by the direct child of the `records` is all. The same is true for `sequence-wrapper-in-root` as well.

4.5. Mapping Expressions

The purpose of DB2XML mapping is to allows users to define the result XML document as a template with placeholders and then generate XML files by applying the template to the appropriate input data streams. Variable parts or placeholders in the template are called **mapping expressions**. Mapping expressions are also used to define generation conditions (see below). Mapping expressions are calculated at the run time, by applying the expression formulas to the current data set. They can be used to define:

- attribute values;
- text node values;
- processing instruction data;
- conditional *`rows-to-use`* for elements;
- generation conditions.

4.5.1. Rows Used by Expression

Expressions do not consume any rows. They are applied to the current row set. Depending on the expression type, it can evaluate using all the available rows, one of them, or none. After expression evaluation, the available row set is not changed.

4.5.2. Expression Examples

`<attribute name="attr" value="'v'"/>` defines an attribute with the constant value *`v`*.

`<text value="Employees.Name"/>` defines a text node which contains the value of the *`Employees.Name`* data field. During generation, the text node will be generated using corresponding value from the current data set.

`<processing-instruction target="role" data="Role" condition="Role"/>` defines a processing instruction with data as the *`Role`* field. During generation, the processing instruction will be generated if *`Role`* is not null or an empty string.

4.5.3. Expressions Kinds

There are two main kinds of expressions: **value expressions** and **conditions**. Value expressions are always evaluated to string to be used in the result XML. Conditions are evaluated to true or false.

4.5.4. Expression Types

Every expression (either value or condition) can be one of the following:

- constant - uses no rows;
- variable - uses the current row;
- function call - uses none, one or any other number of rows from the available row set.

Expression Types and Data Types

Depending on the data type the expression holds, it can be one of the following:

- *`primitive`* expressions;
    - constants or functions;
    - variables of primitive type;
- *`complex`* expressions.

Complex expressions can be:

- *`long character type`* expressions are variables of *`long character type`* or;
- *`binary`* expressions are variables of binary types.

binary expressions are variables of binary types

Constants

Constants in value expressions are defined using *`''`*. If *`''`* are omitted, DB2XML tries to find a data field with that name and treats the expression as variable, if found, and treats as constant, if there is no such data field.

Examples:

- *`<text value="'message'"/>`* - constant text message.
- *`<attribute name="attr" value="Name"/>`* - *`"Name"`* if there is no `Name` data field in the input data stream.

For conditions, there are two constant expressions: `true` or `false`.

Variables

Simply, a variable is a data field from the input data stream.

When a variable is calculated as a value expression, it is copied as a string. Null values are treated as empty strings.

For conditions, the following calculation rules apply:

- If variable is null, than the condition evaluates to false.
- If variable is a Boolean type it value is used for the expression.
- If variable is numeric, if it is 0 (0.0 for floating point numbers) than the expression is evaluated to false, otherwise it is evaluated to true.
- If variable is a string type, and its value is not empty, it evaluates to true. Empty strings are evaluated to false.
- For all other types, the condition is always true unless the corresponding data field is not null.

Examples:

- *`<processing-instruction target="role" data="Role" condition="Role"/<`* - a processing instruction with *`Role`* value expression for data and the same variable for the generation condition.
- *`<attribute name="name" value="Employees.Name"/>`* - the attribute value is generated using the *`Employees.Name`* column from the input table.

Functions

Currently, DB2XML supports the following functions:

- *`bool unchanged(var_name)`* - compares values of *`var_name`* field in the two last rows and returns true, if they are the same, and false otherwise. Returns true if there are less than 2 rows in the current row set.
- *`bool changed(var_name)`* - compares values of var_name field in two last rows and returns true if they are not the same and false otherwise. Returns false if there are less than 2 rows in the current row set.
- *`bool unless(var_name)`* - returns true, if:
  - *`var_name`*data field is null

- or it is boolean and its value is false
- or it is numeric and is 0 (0.0 for floating points)
- or it is string and is empty.

4.6. Conditional Generation

Each attribute and node generation rule may have a condition which is evaluated and the holder attribute or node is generated depending on whether that condition is true or not.

Example: the following mapping rules:

```
<document encoding="UTF-8">
  <element name="company">
    <processing-instruction target="role" data="Role" condition="Role"/>
    <processing-instruction target="role" data="Unknown"
                                 condition="unless(Role)"/>
    <element name="employee">
      <attribute name="employee-id" value="ID"/>
      <element name="name">
        <text value="Name"/>
      </element>
    </element>
  </element>
</document>
```

will generate an XML file which will contain the *`<?role ROLE?>`* processing instruction for rows with non-empty *`Role`* field values (where *`ROLE`* is the value coming from the input data set) and with *`<?role Unknown?>`* for empty *`Role`* values.

Conditions are applicable for all mapping rules (attribute, element, text, processing instruction).

In the ticket files, conditions can be specified as an attribute for a mapping rule (see the example above) or as a child element, as follows:

```
<processing-instruction target="role" data="Role">
  <condition>Role</condition>
</processing-instruction>
```

Here, the condition expression is extracted from the textual content of the `condition` element.

4.7. Namespaces for XML Elements and Attributes

Mapping rules for elements and attributes contain tag/attribute name and namespace definitions using the following properties:

- `name` - local tag name to generate XML element. This property is required;

- `namespace` - namespace for the XML element. If namespace is empty or not specified, the element is generated in the default XML namespace. This property is optional;
- `namespace-prefix` - namespace prefix to use for qualified name construction. If `namespace` is empty or not specified, `namespace-prefix` should be empty. If `namespace-prefix` is not specified or empty, a prefix will be generated for the given namespace.

4.8. Values and Value Converters

Value is a mapping expression which evaluates to string and is used to generate XML attribute values, text node content, and processing instruction data.

Value converters are used to apply specific transformations on evaluated string before final generation.

Value converter can be one of the following types:

- *`plain`* - value is generated as is, without any transformations. It is applicable when the value expression is of a primitive type.
- *`entityfilter`*
  - for attributes - XML special characters are replaced with corresponding entities;
  - for text nodes - < and & special characters are replaced with corresponding entities;
  - for processing instructions - < and > special characters are replaced with corresponding entities.
- *`base64`* - for attributes and text nodes - writes the value as base64 encoded string (it is not applicable for processing instruction data).
- *`cdata`* - generates textual content inside <![CDATA[...]]> (it is applicable for text nodes only).
- *`auto`* - automatic detection of the appropriate converter, which happens using the following rules:
  - For attribute values: When the value expression is of a primitive type, it is treated as *`plain`*, and as *`base64`* otherwise.
  - For text nodes: *`cdata`* is used when the value expression is not primitive and is not binary, *`base64`* is used when it is binary, and *`plain`* is used when is primitive.
  - For processing instructions: It is always treated as plain. Binary value expressions are not supported for processing instruction data.

An empty value converter is the same as *`auto`*.

4.8.1. Value Converter Definition in Ticket File

In the ticket files, values and value converters are defined as attributes for the given mapping rule:

```
<attribute name="Message" value="DataTable.Message" converter="entityfilter"/>
  ...
  <processing-instruction target="process" data="true" converter="plain"/>
  ...
  <text value="Table.Image" converter="base64"/>
```

4.8.2. XML Special Characters and Entities

Table 4.4. XML Special Characters and Corresponding Entities

Character	Entity
&	&amp;
'	&apos;
"	&quot;
<	&lt;
>	&gt;

4.9. Element Mapping Rule

Mapping Rules for elements have the following properties:

- **name and namespace definition properties;**
- **condition**: conditional expression which evaluated before generating the element and its content. If the condition is evaluated to false, nothing is generated and no rows are consumed; If there is no condition, the element is always generated. The root-element mapping rule shouldn't contain a generation condition. This property is optional;
- **rows-to-use**: rows consumed by the given element. For the root element mapping rule, the value of this property should be `all`. This property is optional. Possible values are:
  - *`one`* or omitted - element consumes one row;
  - *`all`* - element consumes all the rows available in the parent context;
  - *`conditional-expression`* - element consumes rows available in the parent context until the condition is true. For grouping, when *`unchanged(Column_name)`* function is used, the input data should be sorted by the given column.

An element mapping rule may optionally contain attributes and child nodes as mapping rules.

There is a special case of element mapping rule: Element mapping rule with no attributes and with a single child text node. In this case, the value of the **rows-to-use** property is ignored. The child text node is generated only once if it does not have a generation condition or the condition evaluates to true.

4.9.1. Ticket Examples For Element Mapping Rules

An element with a single text child:

```
<element name="Name">
  <text value="Table.Name"/>
</element>
```

Conditional element:

```
<element name="Name" condition="Name">
  <text value="Table.Name"/>
</element>
```

Elements from a custom namespace with automatic prefix generation and with fixed prefix:

```
<element name="data" namespace="http://www.example.com/data">
  <element name="data" namespace="http://www.example.com/data"
            namespace-prefix="dn">
    <text value="Table.Data"/>
  </element>
</element>
```

4.10. Attribute Mapping Rule

Attribute mapping rule has the following properties:

- **name and namespace definition properties**;
- **condition**;
- **value**
- **value converter**

4.10.1. Ticket Examples For Attribute Mapping Rules

Examples below show how to define attribute mapping rules in ticket files:

```
<element name="person">
  <attribute name="firstname" value="Table.FirstName"/>
  <attribute name="lastname" value="Table.LastName" condition="Table.LastName"/>
</element>
...
<element name="statements">
  <attribute name="bank" value="'MyBank'"/>
</element>
...
<element name="image">
  <attribute name="image-data" value="Table.Image" converter="base64"/>
</element>
```

4.11. Text Mapping Rule

Text Mapping Rule has the following properties:

- **condition**
- **value**
- **value converter**

4.11.1. Defining Text Mapping Rules in Ticket Files

Examples below show how to define text mapping rules in ticket files:

```
<element name="person-name">
  <text value="'Name:'"/>
  <text value="Users.Name" converter="plain"/>
</element>
...
  <text value="Images.Logo" converter="auto"/>
```

4.12. Processing Instruction Mapping Rule

Processing Instruction mapping rule has the following properties:

- **condition**.
- **target** - XML processing instruction target.
- **data** - value expression to evaluate XML processing instruction data.
- **converter** - value converter for data.

4.12.1. Defining Processing Instruction Mapping Rules in Ticket Files

Examples below show how to define processing instruction mapping rules in ticket files:

```
<processing-instruction target="process" data="true"
                        converter="auto"
                        condition="unless(Messages.Unhandled)"/>
...
<processing-instruction target="xepx-vdpmill-skip-join" data="PS"
                           condition="unless(Accounts.NoHardCopyStatements)"/>
```

Chapter 5. Job Ticket File Format

DB2XML Ticket files describe the data source, output parameters and mapping rules for XML generation. DB2XML GUI Application uses this format to store/load projects.

5.1. Ticket Example

```
<?xml version="1.0" encoding="utf-8"?>
<project version="2.0" xmlns="http://www.renderx.com/DB2XML/job/definition">
  <datasource>
    <database type="System.Data.SqlClient">
      <connection-params>
        <param name="Data Source" value=".\SQLEXPRESS"/>
        <param name="Port" value="1433"/>
        <param name="User ID" value="sa"/>
        <param name="Password" value=""/>
        <param name="Initial Catalog" value="db2xml_TEST"/>
        <param name="Integrated Security" value="False"/>
        <param name="Connect Timeout" value="30"/>
        <param name="User Instance" value="False"/>
      </connection-params>
      <connection-string><![CDATA[]]></connection-string>
      <sqlquery><![CDATA[
        Select * From Accounts
        Join Transactions On Accounts.ACCID=Transactions.AccountID
        Join Customers On Customers.ID=Accounts.CustomerID
        Order By Accounts.ACCID]]></sqlquery>
    </database>
  </datasource>
  <result number-of-records="all" file="Statements.xml">
    <document encoding="UTF-8">
      <element name="statements" rows-to-use="all">
        <element name="statement" rows-to-use="unchanged(Accounts.ACCID)">
          <processing-instruction target="xepx-vdpmill-skip-gen"
                       converter="Auto" data="pdf"
                       condition="unless(Accounts.EMailStatementIsNeeded)"/>
          <processing-instruction target="xepx-vdpmill-skip-join"
                       converter="Auto" data="ps"
                       condition="unless(Accounts.HardStatementIsNeeded)"/>
          <element name="customer" rows-to-use="one">
            <element name="firstname" rows-to-use="one">
              <text converter="Auto" value="Customers.FirstName"/>
            </element>
            <element name="lastname" rows-to-use="one">
```

```
              <text converter="Auto" value="Customers.LastName"/>
            </element>
            <element name="address" rows-to-use="one">
              <element name="postal" rows-to-use="one">
                <text converter="Auto" value="Customers.Address"/>
              </element>
              <element name="email" rows-to-use="one">
                <text converter="Auto" value="Customers.EMail"/>
              </element>
            </element>
            <attribute name="customer-id" converter="Auto"
                          value="Customers.ID"/>
          </element>
          <element name="account" rows-to-use="one">
            <attribute name="account-id" converter="Auto"
                          value="Accounts.ACCID"/>
            <attribute name="amount" converter="Auto"
                          value="Accounts.Amount"/>
            <attribute name="currency" converter="Auto"
                          value="Accounts.Currency"/>
          </element>
          <element name="transactions" rows-to-use="all">
            <element name="transaction" rows-to-use="one">
              <element name="amount" rows-to-use="one">
                <text converter="Auto" value="Transactions.TransAmount"/>
              </element>
              <element name="peer" rows-to-use="one">
                <text converter="Auto" value="Transactions.Peer"/>
              </element>
              <element name="description" rows-to-use="one">
                <text converter="Auto" value="Transactions.Description"/>
              </element>
              <attribute name="transaction-id" converter="Auto"
                            value="Transactions.TransID"/>
              <attribute name="date" converter="Auto"
                            value="Transactions.Date"/>
            </element>
          </element>
        </element>
      </element>
    </document>
  </result>
</project>
```

5.2. Ticket File Format

The following Relax NG Compact schema describes DB2XML Ticket Format:

```
default namespace = "http://www.renderx.com/DB2XML/job/definition"

# ***************************************


start = project

project = element project { version, datasource, result }

version = attribute version { xsd:string {pattern = "[0-9]+\.[0-9]+" }}

datasource = element datasource { database | csv }

parameter = element param { attribute name {xsd:string},
attribute value {xsd:string}
}

#*********** database section ***********

supported.databases = "System.Data.SqlClient" |
"MySql.Data.MySqlClient" |
"Oracle.DataAccess.Client" |
"System.Data.OracleClient"

database = element database {
attribute type { supported.databases },
connection-params, connection-string,
sqlquery
}

connection-params = element connection-params { parameter+ }

connection-string = element connection-string { text }

sqlquery = element sqlquery { text }

#***************************************

#****** CSV Section ********************
```

```
csv = element csv { parameter+ }

#***************************************



#****** Result Section *****************

#---- Common definitions --------
expression.type = xsd:string
condition.type = expression.type

inline-value-converter = "Auto" | "Plain" | "ENTITYFILTER" | "BASE64"
text-value-converter = inline-value-converter | "CDATA"

inline-converter = attribute converter { inline-value-converter }
text-converter = attribute converter { text-value-converter }

condition = attribute condition {condition.type} |
            element condition {condition.type}

value = attribute value {expression.type}

rows-to-use.type = "all" | "one" | condition.type

name = attribute name { xsd:NCName }
ns = attribute namespace { xsd:anyURI }
nsp = attribute namespace-prefix { xsd:NCName }

namespace-definition = ns? | (ns, nsp)?

#--------------------------------



result = element result {
attribute number-of-records { "all" | xsd:positiveInteger },
attribute file {xsd:string},
document }

#***************************************

#****** Mapping ************************

document = element document {
attribute encoding {xsd:string},
pi*,
root-element
}

#---- PI ------------------------
target = attribute target { xsd:NMTOKEN }
data = attribute data { expression.type }

pi = element processing-instruction {
condition?,
inline-converter?,
target,
data
}
#--------------------------------

#---- Text ----------------------

txt = element text {
condition?,
text-converter?,
value
}
#--------------------------------

#---- Attribute -----------------
attr = element attribute {
condition?,
inline-converter?,
name,
namespace-definition,
value
}
#--------------------------------

#---- Root Element --------------
root-element = element element {
name,
namespace-definition,
attribute rows-to-use {"all"},
attr*,
node+
}
#--------------------------------
```

```
#---- Element -------------------
elem = element element {
condition?,
name,
namespace-definition,
attribute rows-to-use { rows-to-use.type },
attr*,
node*
}
#--------------------------------

node = elem | txt | pi

#***************************************
```