Manipulating XML Using Data Integrator

Applies to:
SAP BusinessObjects Data Integrator 11.7.0.0
For more information, visit the Business Objects homepage.

Summary
This white paper shows how to manipulate XML using Data Integrator (DI). The techniques used to manipulate XML are required in various areas in DI, for example when reading and writing XML files, reading and writing XML messages in real-time jobs or when calling web services. This is not a DI tutorial and knowledge of building DI batch jobs is required. However, I have assumed that the audience has no XML or DI real-time knowledge.

This document refers to supplemental files that are available for download. In the following DI examples, all the sample files have been written to local folder -c:\documents. If you are going to follow the examples, place the sample files into a location where your job server can read them, and use that location when building the examples.

Author: Michael Eaton
Company: SAP
Created on: 15 February 2009
# Table of Contents

XML.....................................................................................................................................................................3
- Introduction .....................................................................................................................................................3
- XML Correctness ............................................................................................................................................3
- XML Documents..............................................................................................................................................3
- XML Definitions ...............................................................................................................................................5
  - Document Type Definition..........................................................................................................................5
  - XML Schema Definition ...............................................................................................................................6
- Advantages of XML.........................................................................................................................................7
- Disadvantages of XML....................................................................................................................................7
- Working with XML ...........................................................................................................................................8
  - Reading and writing XML files using DI...........................................................................................................9
    - Importing the XML definition .........................................................................................................................9
    - Reading XML Files........................................................................................................................................13
    - Writing XML files .........................................................................................................................................20
  - Reading and Writing XML from databases ......................................................................................................28
    - Writing XML documents to a relational table .................................................................................................28
    - Reading XML Documents from a Relational Table ......................................................................................31
- Related Content................................................................................................................................................36
- Copyright...........................................................................................................................................................37
XML

Introduction

Extensible markup language (XML) and related technologies are now widely used. Its roots were as a simple mechanism for defining the structure of data, however it has become more powerful and is used as the basis of more complex technologies, for example SOAP web services and XML style sheet transformations.

There are many ways that XML can be used but in this document I am assuming that it is being used to facilitate the sharing of structured data across different information systems. When used for this purpose, there must be an XML definition, this is designed by an XML architect and stored as a DTD or schema; documents are then created and validated using this definition.

In the following sections I create the definition of a simple XML that is used to define documents that will contain postcodes and addresses. These documents and definitions are then used to build some DI examples.

The following websites contain more detail about XML:

- [http://en.wikipedia.org/wiki/XML](http://en.wikipedia.org/wiki/XML) (some of the following is from this definition)
- [http://www.w3.org/XML](http://www.w3.org/XML)
- [http://www.xml.org](http://www.xml.org)

XML Correctness

There are two levels of correctness of an XML document:

- Well formed. The basic structure of the document must be correct. Each opening tag must have a closing tag in the correct position.
- Valid. The document must conform to the XML definition (stored in a DTD or schema)

Software known as a validating parser is used to test for XML correctness.

XML Documents

XML documents are text based and the data within them must be marked-up (enclosed) in tags. Tag names are case sensitive. When delineating data, it must be enclosed with an opening tag and closing tag (that uses a forward slash). For example:

```
<phoneNumber>01234 123456</phoneNumber>
```

The example above shows an XML element called `phoneNumber`. Tags may be nested, but they cannot overlap. The following is not valid:

```
<name><first>Michael</first></name>
```

It should be

```
<name><first>Michael</first></name>
```

White space between tags in a document is ignored and tabs or spaces are often inserted to make documents more readable. For example:

```
<name>
  <first>Michael</first>
</name>
```

Empty elements can use the special self-closing tag. The following are logically the same:

```
<name/>
```

```
<name></name>
```
Comments can be inserted into a document as follows:

<!-- Created by Michael Eaton. -->

Elements can have attributes to describe them, for example:

<person age="36" eyes="blue">Michael</person>

An alternative representation of the data above is

<person>
  <age>36</age>
  <eyes>blue</eyes>
  <name>Michael</name>
</person>

An XML declaration is optional, but normally included at the start of the document:

<?xml version="1.0" encoding="UTF-8"?>

As long as only a well-formed document is required, XML is a generic framework for storing any amount of text or any data whose structure can be represented as a tree. The only indispensable syntactical requirement is that the document has exactly one root element (alternatively called the document element). The following is a well-formed XML document containing data for postcodes and addresses; the root element is postcodeAddress:

<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by Michael Eaton. -->
<postcodeAddress>
  <postcodeGroup postcode="SA019TB">
    <address>Flat 1, Block C, Some Road</address>
    <address>Flat 2, Block C, Some Road</address>
    <address>Flat 3, Block C, Some Road</address>
  </postcodeGroup>
  <postcodeGroup postcode="DE150LQ">
    <address>Eaton Hall, Eaton Estate</address>
    <address>The Gardeners Cottage, Eaton Estate</address>
  </postcodeGroup>
</postcodeAddress>

This XML document is in the sample file postcodeAddress.xml. The XML document above is well formed, but is it valid? The answer is “uncertain”. At this stage we have no way of checking if the structure or data types are correct, for that we require an XML definition.
XML Definitions

To ensure the validity of an XML document for some domain, we require a definition. They are typically stored as a document type definition (DTD) or as an XML schema definition (XSD). The main advantage of XSDs is that they provide a lot more flexibility over DTDs. For example, in an XML schema you can specify that an element must be an integer and that its value must lie between two bounds, you can define collections of elements, and you can define strings whose length lies between two values.

An XML document that complies with a particular schema or DTD, in addition to being well formed, is said to be valid.

Designing and creating real world definition documents (now normally using an XSD) is an involved task, so I will not explain this in any detail, I will just provide example definitions. Definitions can be published to a web server for wide access or to a file system in a specific domain. If a reference to definition is embedded in an XML document, then the definition must be accessible from all locations where that document is to be used.

Document Type Definition

The following DTD specifies a definition for the postcode and address data. It is in the sample file postcodeAddress.dtd.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD written by Michael Eaton -->
<!ELEMENT postcodeAddress (postcodeGroup*)>
<!ELEMENT postcodeGroup (address+)>
<!ELEMENT address (#PCDATA)>
<!ATTLIST postcodeGroup
  postcode CDATA #REQUIRED>

This definition specifies that there can be a sequence of zero or more postcodeGroup elements, and for each one there must be 1 or more address elements. In a DTD an asterisk indicates zero or more and the plus indicates one or more. We cannot specify data-types in a DTD. Within the DTD #PCDATA means parsed character data and CDATA means character data.

Once we have a definition we can use it to validate an XML document using a validating parser (XML Spy has this functionality). To remove any ambiguity when using XML we can place a reference to the definition in an XML document. The following example document (postcodeAddress_usingDTD.xml) demonstrates this.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by Michael Eaton. -->
<!DOCTYPE postcodeAddress SYSTEM "./postcodeAddress.dtd">
<postcodeAddress>
  <postcodeGroup postcode="SA019TB">
    <address>Flat 1, Block C, Some Road</address>
    <address>Flat 2, Block C, Some Road</address>
    <address>Flat 3, Block C, Some Road</address>
  </postcodeGroup>
  <postcodeGroup postcode="DE150LQ">
    <address>Eaton Hall, Eaton Estate</address>
    <address>The Gardeners Cottage, Eaton Estate</address>
  </postcodeGroup>
</postcodeAddress>
```
XML Schema Definition

The following XSD specifies an alternative definition for the postcode and address data. It is in the sample file postcodeAddress.xsd.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- XSD written by Michael Eaton -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xs:element name="address">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:maxLength value="50"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>

  <xs:element name="postcodeAddress">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="postcodeGroup" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="postcodeGroup">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="address" maxOccurs="unbounded"/>
        <xs:attribute name="postcode" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:maxLength value="7"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Looking at this definition we can see that data-types can be specified, and in the example, a maximum length of a string. We can also specify how many instances of an element must exist, using `minOccurs` and `maxOccurs`. 
The following example (postcodeAddress_usingXSD.xml) demonstrates how to specify a reference to an XSD.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by Michael Eaton. -->
<postcodeAddress xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=".\postcodeAddress.xsd">
  <postcodeGroup postcode="SA019TB">
    <address>Flat 1, Block C, Some Road</address>
    <address>Flat 2, Block C, Some Road</address>
    <address>Flat 3, Block C, Some Road</address>
  </postcodeGroup>
  <postcodeGroup postcode="DE150LQ">
    <address>Eaton Hall, Eaton Estate</address>
    <address>The Gardeners Cottage, Eaton Estate</address>
  </postcodeGroup>
</postcodeAddress>
```

**Advantages of XML**

- It is text-based, so it can be read easily.
- It supports Unicode, allowing almost any information in any written human language to be communicated.
- It can represent the most general data structures: records, lists and trees.
- Its self-documenting format describes structure and field names as well as specific values.
- XML is heavily used as a format for document storage and processing, both online and offline.
- It is based on international standards.
- It can be updated incrementally.
- It allows validation using schema languages such as XSD, which makes system construction easier as common components can perform the validation.
- The hierarchical structure is suitable for most (but not all) types of documents.
- It manifests as plain text files, which are less restrictive than other proprietary document formats.
- It is platform-independent, thus relatively immune to changes in technology.
- Forward and backward compatibility are relatively easy to maintain despite changes in DTD or Schema.
- There is extensive experience and software available.
- An element fragment (any opening tag to its closing tag) of a well-formed XML document is also a well-formed XML document.

**Disadvantages of XML**

- XML syntax is redundant or large compared to binary representations of similar data.
- The redundancy may affect application efficiency through higher storage, transmission and processing demands.
- XML syntax is verbose relative to other alternative "text-based" data transmission formats.
- The hierarchical model for representation is limited in comparison to the relational model or an object oriented graph.
- Expressing overlapping (non-hierarchical) node relationships requires extra effort.
- XML namespaces are problematic to use.
- XML is commonly depicted as "self-documenting" but this depiction ignores critical ambiguities.
Working with XML

Realistically, when working with anything other than the most basic XML, specialized editors, DTD and schema development tools, parsers and tools for validating are required. There are many open source and commercial tools available. In the creation of this white paper, I used Altova XML Spy. It has many useful features, for example it can display graphical representations of schemas and documents (shown below), and can generate sample documents from schemas.

Generated with XMLSpy Schema Editor www.xmlspy.com
Reading and writing XML files using DI

Working with XML is not as straightforward as working with the usual flat structures, such as relational tables or flat files. It requires an understanding of how to work with DI’s nested relational data model (NRDM), which is used by DI internally to represent XML documents. This section demonstrates how to read and write XML documents for a specific schema in a batch job.

Importing the XML definition

When working with XML in DI, the first thing to do is to import the metadata for the XML definition. DI supports both DTDs and XSDs. In the following examples, the definition is imported from a local drive, and but in real-world scenarios, the location of the definition would be published within the domain where it would be used, and referenced by a URL or UNC.

1. To import a DTD, create a new DTD within DI, this creates a new object to store the metadata for the DTD.

2. When importing the DTD (see figure 1) specify a name for it within DI (you could use the DTD name), the location of the DTD, and the root element name.

3. Click OK to import the DTD.

![Figure 1 - importing a DTD](image-url)
DI imports the definition and it can be seen in the local repository under the formats tab.

4. Double-click this DTD to display the NRDM representation of the DTD (Figure 3). Note that the data types are varchar(1024) because DTDs do not support data types and DI is using varchar(1024) as a default.

5. As we will not be using this DTD in the following examples, delete the definition.
6. To import an XSD, first create a new XSD object.

7. Specify the name, XSD location, and root element name (this example uses the sample postcode XSD). Once we have imported the definition it can be viewed in the formats tab of the local object library under XML Schemas.
8. Double-click the schema to display the NRDM structure for it (Figure 4). Notice that the data-types have been imported from the schema and they are much more suitable. From this point forward, this document will use this schema.

<table>
<thead>
<tr>
<th>Schema: postcodeAddress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
</tr>
<tr>
<td>postcodeGroup</td>
</tr>
<tr>
<td>postcode</td>
</tr>
<tr>
<td>postcodeGroup_nt_1</td>
</tr>
<tr>
<td>address</td>
</tr>
</tbody>
</table>

*Figure 4 - NRDM for schema*
Reading XML Files

In this example we will read the sample postcode XML file (postcodeAddress.xml) and write the data to two relational tables - postcode and address.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by Michael Eaton. -->
<postcodeAddress>
  <postcodeGroup postcode="SA019TB">
    <address>Flat 1, Block C, Some Road</address>
    <address>Flat 2, Block C, Some Road</address>
    <address>Flat 3, Block C, Some Road</address>
  </postcodeGroup>
  <postcodeGroup postcode="DE150LQ">
    <address>Eaton Hall, Eaton Estate</address>
    <address>The Gardeners Cottage, Eaton Estate</address>
  </postcodeGroup>
</postcodeAddress>
```

1. Create a new batch job and a new dataflow. My job is called JOB_ReadWrite_XML (as we will add writing to it in the next section) and the new dataflow is called DF_Read_XML.

2. In the designer drag the postcodeAddress XML schema object from the local object library onto the dataflow workspace. A menu appears as shown in figure 5.

```
Figure 5 - Schema menu
```

3. Select the Make XML File Source option. A XML file source is added to the dataflow.

postcodeAddress

4. Open the editor for the XML file and specify the runtime location of the XML file.
5. Next, add a query which we will use to un-nest the data.

6. Open the query editor and drag the postcodeGroup node from the Schema In to the Schema Out as shown below. Now we have an NRDM of the postcode data. However, to write the data to relational tables, we must flatten the data using DI's un-nest feature.

7. Right-click the **postcodeGroup** node in the Schema Out and select un-nest, as shown below. You should see the icon change to include a small arrow to indicate that the element will be un-nested.
We will also need to flatten the postcodeGroup_nt_1 (a DI generated nested table to hold multiple addresses for each postcode). Before it can be un-nested we must make the nested structure current. This is one of the major differences when working with NRDM, each level of nesting is handled with a separate `from` clause, `where` clause, `order by`, etc. This is more relevant when nesting data rather than un-nesting and is explained in the section covering writing XML files.

8. To make the postcodeGroup_nt_1 current, right-click it and select **Make Current** (alternatively double-click the node).
9. Once it is selected, un-nest it as above. Schema Out should now look like Figure 6.

![Figure 6 - Un-nested Schema Out](image)

10. Add a template table to the dataflow and connect it to the query.

![Diagram](image)

11. To see the result of the un-nesting operations, open the editor for the template table (Figure 7).

![Figure 7 – Un-nested postcode data](image)
12. Run the job to un-nest the postcode data and write it into the XML_UNNEST table. The results are shown in Figure 8.

**Figure 8 - Un-nested postcode data**

During the un-nest operation, DI replicates the postcode for every relevant address. This is a very useful feature, because when we flatten a hierarchy of data, we generally need to retain the relationships that are implicit by the position of data in a tree. In our example, an address is related to a postcode by its position “below” a specific postcode. DI helps us to maintain these relationships.

A more realistic example would be to generate a surrogate key for the postcode and to assign it to the address, then write the postcodes to one table and the addresses to another. This is demonstrated below.

1. Remove the query and template table and add three new queries and two new template tables for postcodes and addresses (I called mine demo_postcode and demo_address). The dataflow should be structured as shown in Figure 9.

**Figure 9 - dataflow to split postcodes and addresses**

2. Open the query editor for add_id and drag the postcodeGroup node from the Schema In to the Schema Out as shown below.
3. Make the postgroupGroup node current and un-nest it (as explained above).

4. We now need to add a new column for the surrogate key at the same level as the postcode element. When adding columns it is important to ensure that they are at the correct level in the hierarchy. Make the postcodeGroup node current then right-click on the postcodeGroup node and add a new output column.
5. Insert it below the existing node and call it postcode_id (with int data-type).

The new column might look slightly out of place, but that does not matter as long as it is at the correct level in the hierarchy. The mapping for the new column should be `gen_row_num()` to generate the surrogate key.

6. Open the editor for the get_postcode query. Drag the postcode_id and postcode columns from Schema In to Schema Out.

7. Open the editor for the get_address query and drag the postcode_id column and the postcodeGroup_nt_1 schema from Schema In to Schema Out.
8. Make the postcodeGroup_nt_1 node current and un-nest it.

<table>
<thead>
<tr>
<th>Schema In:</th>
<th>Schema Out:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

The dataflow is now complete. When we execute it we should obtain the results below. As you can see, the generated postcode IDs have been correctly assigned to each address.

Writing XML files

In this section we will construct an XML document containing the postcode data and write it to a file. The XML definition (XSD) we will use is slightly different to the one used above. It has additional elements to store some document related data. The schema and its graphical representation are shown below.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v2004 rel. 2 U (http://www.xmlspy.com) -->
<!-- XSD written by Michael Eaton -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified">
  <xs:element name="address">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:maxLength value="50"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:element name="postcodeAddress">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="postcodeGroup" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="documentDetail">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="createdBy">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:maxLength value="30"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

© 2009 SAP AG
<xs:element name="createDate" type="xs:date"/>
<xs:element name="createComment">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:maxLength value="50"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

postcodeAddress —- postcodeGroup
                  —- address
                              0..∞

documentDetail —- postcodeGroup
                          —- address
                                      1..∞

createdBy

createDate

createComment
This schema is in the sample file `postcodeAddress2.xsd`.

1. Import this schema using the same mechanism as above (Project|New|XML Schema...) and call it `postcodeAddress2`. The NRDM for the schema is shown below.

<table>
<thead>
<tr>
<th>Schema:</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>postcodeAddress2</td>
<td></td>
</tr>
<tr>
<td>postcodeGroup</td>
<td></td>
</tr>
<tr>
<td></td>
<td>postcode varchar(7)</td>
</tr>
<tr>
<td></td>
<td>postcodeGroup_nt_1</td>
</tr>
<tr>
<td></td>
<td>address varchar(50)</td>
</tr>
<tr>
<td>documentDetail</td>
<td></td>
</tr>
<tr>
<td></td>
<td>createdBy varchar(30)</td>
</tr>
<tr>
<td></td>
<td>createDate date</td>
</tr>
<tr>
<td></td>
<td>createComment varchar(50)</td>
</tr>
</tbody>
</table>

2. Add a new dataflow to the existing job, I called my dataflow `DF_WriteXML`.

3. Drag the `postcodeAddress2` schema into the dataflow workspace and select `Make XML File Target`.

4. Open the editor for the XML file and specify a target filename. Make sure that Delete and re-create file are selected.
5. Add the two previously created template tables and a row generation as sources, together with a query and connect them as shown below. The row generation row count must be one (the default).

6. Open the editor for the query.

Each of the nested nodes (nest_data, postcodeGroup, postcodeGroup_nt_1 and documentDetail) must be mapped individually.
7. Make the top level node and nest_data current by right-clicking each and clicking Make Current.

Note that the from clause contains all three sources, we can remove DEMO_POSTCODE and DEMO_ADDRESS as they are not required for this node. We will use the row generation to ensure that we have a root node in our XML document. (We could have left the two tables in this top level node but it can be confusing. Any time a new source is added to the query, it will appear in the from list of the top level node.)

8. Make the postcodeGroup node current and note that the from clause is now empty. Each nested node gets a separate set of from, Outer Join, Where Group By, etc. It is actually a whole new query for this level in the hierarchy.
To map any elements in this level we need to add a source to the from list. We know that in the XML document we require data for postcodes for this node.

9. Drag the DEMO_POSTCODE source from the Schema In section to the from list. We can then map the postcode element to the POSTCODE column.

10. Make the postcodeGroup_nt_1 node current. Note that the from list is again empty.

11. We need address data for this node; therefore, add DEMO_ADDRESS to the from list and then map the address element to the ADDRESS column.
12. You see all the addresses appearing for all the postcodes. Filter the addresses so that only the relevant addresses appear under each postcode by using a WHERE clause: `DEMO_ADDRESS.POSTCODE_ID = DEMO_POSTCODE.POSTCODE_ID`.

Even though `DEMO_POSTCODE` source does not appear in the `from` list for this node, we can still reference it in the where clause because it is in a `from` list for a node above this one in the hierarchy. A common mistake would be to add the `DEMO_POSTCODE` source to the `from` list for this node. That would cause too many elements to be written to the XML document.

The postcode and address sections are dynamic and can contain multiple sets of data based on the data in our source tables; however the documentDetail node will only ever occur once in the XML document.

13. Use the row generation source by adding it to the `from` list for this node.

14. Make the documentDetail node current and add the row generation to the `from` list. The createdBy, createDate and createComment elements can now be mapped.
The dataflow is now complete. When we execute the job DI writes the following XML document to file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- BusinessObjects Data Integrator generated XML -->
<!-- 2007-12-05.14:02:34(299,990)[1] -->
<postcodeAddress xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\Documents\postcodeAddress2.xsd">
  <postcodeGroup postcode="SA019TB">
    <address>Flat 1, Block C, Some Road</address>
    <address>Flat 2, Block C, Some Road</address>
    <address>Flat 3, Block C, Some Road</address>
  </postcodeGroup>
  <postcodeGroup postcode="DE150LQ">
    <address>Eaton Hall, Eaton Estate</address>
    <address>The Gardeners Cottage, Eaton Estate</address>
  </postcodeGroup>
  <documentDetail>
    <createdBy>Michael Eaton</createdBy>
    <createDate>2007-12-05</createDate>
    <createComment>Example document for white paper</createComment>
  </documentDetail>
</postcodeAddress>
```

This XML document is in the sample file `postcodeAddress2.xml`. 
Reading and Writing XML from databases

In the previous section the XML documents that were manipulated were stored as text files. It is also possible to manipulate XML that is stored as a text string in a column in a relational table using DI.

Writing XML documents to a relational table

In this example we will replicate the dataflow above (DF_WriteXML) and modify it to store the XML it generates into a column in a template table.

1. First create a new job and add the replicated dataflow (for example, JOB_ReadWrite_DB_XML and the renamed the replicated dataflow to DF_Write_DB_XML).

2. Open the dataflow, delete the target XML file and add two new queries and a template table as shown below.
3. Open the editor for the **add_new_level** query.

4. Drag the **nest_data** node from **Schema in** and drop it on the **add_new_level** node in **Schema Out**.

5. Open the editor for the **convert_to_varchar** query and add an output column called **xml_data** with a datatype of varchar(1000). If larger XML documents are expected, DI does support Oracle **CLOB** and SQL Server **TEXT** columns as a **LONG** data-type, and provides two functions to convert between **LONG** and **VARCHAR**, **long_to_varchar** and **varchar_to_long**.

6. For the mapping of the new column click the **Functions** button, select **load_to_xml** function from the conversion category, and then click **Next**.
7. Specify the function input parameters as shown below.

![Define Input Parameter(s)](image)

8. Click the **Finish** button to complete the mapping: it should be `load_to_xml(nest_data, 'postcodeAddress2', 0, '', '', 1, 1000)`. This dataflow is now complete. If we browse the data in the template table once the job has executed, we will see that DI has written the XML as a string to the table as shown below. (Right-click and select **View cell** to see all the content.)

```
<tlm version="1.0" encoding="UTF-8">
<postcodeAddress xmlns="http://www.nhs.org/2001/2/MLSAnonymised">
<auxName>postCodeAddress</auxName>
<postCodeAddressGroup postCode="SA109TH">
<address>Flat 1, Block C, Some Roads</address>
</postCodeAddressGroup>
<postCodeAddressGroup postCode="DE1591G">
<address>Eaton Hall, Eaton Estates</address>
</postCodeAddressGroup>
<postCodeAddressGroup postCode="SA109TH">
<address>Flat 2, Block C, Some Roads</address>
</postCodeAddressGroup>
<postCodeAddressGroup postCode="DE1591G">
<address>The Gardens, Eaton Estates</address>
</postCodeAddressGroup>
<document>Example document for white paper</document>
</tlm>
```
Reading XML Documents from a Relational Table

The above example wrote an XML document into a relational table called DEMO_DB_XML. This section will read the document and write it to a text file using a template XML object.

1. Add a new dataflow to job, I called mine DF_Read_DB_XML.

2. Add the template table used in the above section and a query to the dataflow.

3. Open the editor for the extract_xml query and right-click the extract_xml node in the Schema Out.
4. Click the **New Function Call** option from the pop-up menu as shown below.

5. Select the **extract_from_xml** function from the Conversion Functions and click **Next**.
6. Complete the function parameters as shown below and click **Next**.

![Function parameters screenshot]

7. Move the `postcodeGroup` and `documentDetail` parameters to **Selected output** section and click **Finish**.

![Selected output parameters screenshot]
The query should now look like the following screenshot. The output of the `extract_from_xml` function is an NRDM that matches the postcodeAddress2 schema.

8. Add a template XML object to the dataflow (it can be found on the Palette toolbar).
9. Open the editor for the XML Template and specify a filename.

10. Connect the query to XML Template.
The dataflow is now complete. When we execute it, DI reads the text from the table and converts it into the NRDM representation. The output file created by the XML template object should look like the following XML.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- BusinessObjects Data Integrator generated XML -->
<!-- 2007-12-21 16:12:49(279,279)[1] -->
<read_xml_demo>
  <postcodeGroup>
    <postcode>SA019TB</postcode>
    <postcodeGroup_nt_1>
      <address>Flat 1, Block C, Some Road</address>
    </postcodeGroup_nt_1>
    <postcodeGroup_nt_1>
      <address>Flat 2, Block C, Some Road</address>
    </postcodeGroup_nt_1>
    <postcodeGroup_nt_1>
      <address>Flat 3, Block C, Some Road</address>
    </postcodeGroup_nt_1>
  </postcodeGroup>
  <postcodeGroup>
    <postcode>DE150LQ</postcode>
    <postcodeGroup_nt_1>
      <address>Eaton Hall, Eaton Estate</address>
    </postcodeGroup_nt_1>
    <postcodeGroup_nt_1>
      <address>The Gardeners Cottage, Eaton Estate</address>
    </postcodeGroup_nt_1>
  </postcodeGroup>
  <documentDetail>
    <createdBy>Michael Eaton</createdBy>
    <createDate>2007.12.21</createDate>
    <createComment>Example document for white paper</createComment>
  </documentDetail>
</read_xml_demo>
```

This XML is in the sample file `read_xml_demo.xml`. Note that the structure does not exactly match that of the original XML document, this is because the XML Template object has no knowledge of the required XML schema for this data. This is an important point, only use the XML Template object for debugging or basic testing.
Related Content

BusinessObjects Information Management Community

Supplemental files for this article
Copyright

© Copyright 2009 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors. Microsoft, Windows, Excel, Outlook, and PowerPoint are registered trademarks of Microsoft Corporation.

IBM, DB2, DB2 Universal Database, System i, System i5, System p, System x, System z, System z10, System z9, z10, z9, iSeries, pSeries, xSeries, zSeries, eServer, zVM, z/OS, i5/OS, S/390, OS/390, OS/400, AS/400, S/390 Parallel Enterprise Server, PowerVM, Power Architecture, POWER6+, POWER6, POWER5+, POWER5, POWER, OpenPower, PowerPC, BatchPipes, BladeCenter, System Storage, GPFS, HACMP, RETAIN, DB2 Connect, RACF, Redbooks, OS/2, Parallel Sysplex, MVS/ESA, AIX, Intelligent Miner, WebSphere, Netfinity, Tivoli and Informix are trademarks or registered trademarks of IBM Corporation.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Adobe, the Adobe logo, Acrobat, PostScript, and Reader are either trademarks or registered trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Oracle is a registered trademark of Oracle Corporation.

UNIX, X/Open, OSF/1, and Motif are registered trademarks of the Open Group.

Citrix, ICA, Program Neighborhood, MetaFrame, WinFrame, VideoFrame, and MultiWin are trademarks or registered trademarks of Citrix Systems, Inc.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

Java is a registered trademark of Sun Microsystems, Inc.

JavaScript is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, R/3, SAP NetWeaver, Duet, PartnerEdge, ByDesign, SAP Business ByDesign, and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and other countries.

Business Objects and the Business Objects logo, BusinessObjects, Crystal Reports, Crystal Decisions, Web Intelligence, Xcelsius, and other Business Objects products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of Business Objects S.A. in the United States and in other countries. Business Objects is an SAP company.

All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

These materials are subject to change without notice. These materials are provided by SAP AG and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.