# CNS_Logo_2012_wURLGENERIC XML PARSER Documentation

# 2015.11.10

## GENERIC XML PARSER Documentation

Robert Light & Daniel Halsey

Indiana University

## INTRODUCTION

The generic XML parser is designed to process XML records into SQL script that can then be uploaded into a relational database. The parser converts XML into a series of insert statements based on a configuration file that specifies the tables and columns that are desired by the user.

## CONFIGURATION OPTIONS

There are a number of configuration options that will need to be set in order to run the parser.

**Input Settings**

One of these must be set to indicate which files to process.

**-f --file:** Defines a single file to parse.

**-d --directory:** Defines a directory to parse. All files ending in “.xml” within this directory will be parsed. Subdirectories will NOT be parsed.

**Configuration File Settings**

**-c --config:** Defines the configuration file with the map from XML to schema. This must be present. Details on the construction of this file are below.

**-t --template:** This defines a template file which can be used to create wrappers around the insert statements. This is optional. Details on the construction of this file are below.

**-l --file\_number:** This defines a CSV file with a mapping from file name to file number. This is optional. File numbers are often helpful for maintaining the most current version of a record. Details on the construction of this file are below.

**XML Configuration Settings**

These settings define the basic information that the parser needs to define records.

**-p --parent:** The name of the parent tag of the collection of records to be processed. In many cases, this is the root tag of the entire file, though if only a portion of the file is being processed, this can be defined as a path. (for example: “GreaterFile/CitationSet”). If each file consists of only a single record, this can be the same as the record tag below, but is required.

**-r --record:** The name of the root tag that indicates each specific record to be parsed. “MedlineCitation”, “ClinicalTrial”, etc. All data contained within this tag and its children will be presumed to be part of the same document. These must be direct children of the parent tag/path. This setting is required.

**-i --identifier:** The name of the tag whose value provides the unique identifier for the record. If this is a child of the record tag, the name of the tag is sufficient. Otherwise, give the path beyond the record tag. This setting is required.

Example:

<MedlineCitationSet>

<MedlineCitation>

…

<Identifiers>

<PMID>12345</PMID>

…

</Identifiers>

…

</MedlineCitation>

<MedlineCitation>

…

<Identifiers>

<PMID>11111</PMID>

…

</Identifiers>

…

</MedlineCitation>

…

</MedlineCitationSet>

For this set:

-p would be MedlineCitationSet

-r would be MedlineCitation

-i would be Identifiers/PMID

Other settings

**-n --namespace:** This setting can be used if the XML has a defined namespace. Currently XML with only one namespace can be managed by the parser. This setting is optional.

**-s --single\_trans:** If True, this setting will place a wrapper around each file creating a single transaction for that file. This may help performance in some settings where statements are otherwise automatically committed. This cannot be used if the template includes a transaction statement.

## BUILDING THE CONFIGURATION FILES

**Schema Configuration**

The Schema Configuration file tells the parser what sort of database to build out of the XML.

Consider the following example XML that we want to convert into a database:

<People>

<Person name=”Joe”>

<Emp\_Id>12435</Emp\_Id>

<State>Indiana</State>

<Color>Red</Color>

<Car color=”Blue”>Ford</Car>

<Car color=”White”>Nissan</Car>

<Person>

<Person name=”Amy”>

<Emp\_Id>12435</Emp\_Id>

…and many more people after that

</People>

where we know that every person has a unique Emp\_Id.

Ideally, a schema should be designed from a DTD or XSD that clearly defines every acceptable tag within the XML schema, but we will assume that the data you see here is a complete record.

With this in mind, we design a database schema and create the empty database on our DB server.

Now we need to know our settings. The parent is the <People> tag. This is the collection of records. The record tag is <Person>. The person is our record of interest and everything within that tag refers to the same person. The identifier tag is <Emp\_Id>. Our list might include a dozen Joe’s, but only one 12345.

Now we need a configuration file to tell the parser where to put each element of the XML. Here’s what that looks like

<People>

<Person table=”employee\_list” file\_number=”employee\_list:file\_number” name=”employee\_list:name”>

<Emp\_Id></Emp\_Id>

<State>employee\_list:state</State>

<Color>employee\_list:color</Color>

<Car table=”emp\_cars” ctr\_id=”emp\_cars:car\_ctr” color=”emp\_cars:color”>emp\_cars:car</Car>

</Person>

</People>

<People> is our root. We don’t have to do anything with it. It defines the section of the file that we’re interested in. If there was more to this XML file outside of <People>…</People>, it would be ignored. Starting with person, we define the table that matches to the record. This first table, which we name “employee\_list” will be central to the database. We also being assigning columns with the name attribute, which we assign to “employee\_list:name”. We can assign any attribute to a column on a table that has been defined by that tag or one of its ancestors.

We can do the same with values or the values of children. We assign the value of State to “employee\_list:state”. Note that this requires some knowledge of the data. If we tried to do the same thing for Car, we’re have a big problem, since there is more than one value to write.

The Car tag is part of a one-to-many relationship, so it requires a new table. We define this by creating a new table attribute, giving the name of the new table “emp\_cars”, but we also need a counter, which we define with the ctr\_id “emp\_cars:car\_ctr”

Every new table after the first one MUST have a ctr\_id defined. The identifier serves as the counter for the first table. All other counters will be sequential, counting up from 1 within their context. In other words, if Amy has cars, they’ll start counting from 1 for her.

Now, each instance of the <Car> tag will create a new row in the emp\_cars table. Within that tag, we can assign attributes and values to that row, so we assign the color attribute and the value to columns in the emp\_cars table.

There is one more special attribute in the config file, the file\_number. This allows the user to insert a file\_number into a table, using the File Number Index explained below. This can be very helpful in dealing with records that may be overwritten later in a collection.

Note that we don’t do anything with Emp\_Id. Since we named it the identifier, it automatically gets added to every table as the id column. We could reassign it to a second column within the table if we wanted, but that would be redundant.

Here is the output when the XML for Joe above is run through the config file.

INSERT INTO `employee\_list` (`id`, `file\_number`, `name`, `state`, `color`) VALUES (12345, 15, ‘Joe’, ‘Indiana’, ‘Red’);

INSERT INTO `emp\_cars` (`id`, `car\_ctr`, `car`, `color`) VALUES (12345, 1, ‘Ford’, ‘Blue’);

INSERT INTO `emp\_cars` (`id`, `car\_ctr`, `car`, `color`) VALUES (12345, 2, ‘Nissan’, ‘White’);

**Template File**

The template file is designed to give a wrapper for the INSERT statements generated by the parser. This template is applied once per record.

The template recognizes three variables for insertion into the template.

**%data** – The series of INSERT statements generated for that record. It is best to always put this on its own line in the template file.

**%file\_number** – The file number gathered from the File Number Index.

**%id** – The identifier for the record.

So a simple template designed to only keep the most current version of a record may look like this:

USE employee\_db;

BEGIN

DELETE FROM `employee\_list` WHERE `id` = %id AND file\_number <= %file\_number;

%data

COMMIT;

Applying this to our output from above, we might get this:

USE employee\_db;

BEGIN

DELETE FROM `employee\_list` WHERE `id` = 12345 AND file\_number <= 15;

INSERT INTO `employee\_list` (`id`, `file\_number`, `name`, `state`, `color`) VALUES (12345, 15, ‘Joe’, ‘Indiana’, ‘Red’);

INSERT INTO `emp\_cars` (`id`, `car\_ctr`, `car`, `color`) VALUES (12345, 1, ‘Ford’, ‘Blue’);

INSERT INTO `emp\_cars` (`id`, `car\_ctr`, `car`, `color`) VALUES (12345, 2, ‘Nissan’, ‘White’);

COMMIT;

Assuming that primary and foreign keys are set up within the database and are set to cascade on deletion, then this script should do the following:

If there were no older data: Write the new data.

If there were older data: Delete that and then write in the new data.

If there were NEWER data: Fail on the first insert and rollback, leaving the more current data intact.

Exact syntax will vary by DBMS.

**File Number Index**

This is the most straightforward of the files to build. It is a simple csv file of the format:

name,file\_number

filename1,number

filename2,number

…

Numbers do not need to be sequential or even unique, to allow maximum flexibility for the user’s purposes. The header is not required.