

# xml2db

Matt Skinner — skinner.m.c@gmail.com

April 2, 2010

## 1 Introduction

The xml2db tool aims to ease the conversion process of XML data into a Relational Database System. An XML file contains a single record with a minimal amount of duplicate nodes, one with multiple records (XML database) with a high amount of duplicated nodes, or a combination. As of now xml2db only handles the one record scenario, and unlike other xml importing tools, xml2db assumes the database already exists and will not create or alter database structure.

Xml2db does not work without some configuration because it knows nothing about the database or about the XML. It must be told about the relationship between the XML and the database via a mapping file. To construct a mapping file one must have knowledge of the database and all its fields, knowledge of what each text node in the XML describes or is used for. Then the task is to construct the mapping between the XML text nodes and the fields in the database, and test the mapping for correctness.

## 2 Constructing the Mapping

Xml2db does not know about the relationships between the text nodes of the XML and the fields in the database. To tell the tool about such relationships, a mapping file must be constructed. The tool interprets the file linearly, building a row of data for the database as it appears in the mapping file. Multiple inserts into one or more table is possible, but for purposes of describing the map file we'll require that one XML file is one "record" spread out across multiple tables. Importing XML files with multiple records (XML databases) are not yet supported.

A map file is an XML file specifying a set of mappings to a particular database field (e.g. table.field) from various datasources. Each map *always* describes a mapping from various datasources to a single database field. When a different table name or a duplicate field is encountered the tool ends the construction of the current row and begins a new one; every row to be inserted in a table must have its mappings listed consecutively in the map file, repeating fields or changing rows ends the building of the current row and begins the next. Each map definition must map to an existing database field. The tool will abort execution if such field or table does not exist.

A map file consists of a set of mappings that map various data sources to a particular field in a table. Sources of data can be one, and only one of the following:

1. A static value
2. A xPath 1.0 expression
3. A prompted value
4. A runtime constant

## 2.1 Mapping File

The general structure of a mapping file is shown below:

```
1 <mapset>
  <map>
3     <dbfield>table.field1</dbfield>
  </map>
5   <map>
     <dbfield>table.field2</dbfield>
7   </map>
</mapset>
```

The above mapping file has root nodes `<mapset>` with children `<map>` specifying all the mappings to the database fields. It should be obvious what database field is being mapped to and what is being mapped from using XML markup. The above example is incomplete as it lacks anything to map from. In fact, the tool will complain and abort execution. To correct this we must define what is being mapped from.

## 2.2 `<value>`

The `<value>` node describes a static value to map to a particular field in the database, useful in situations where all the XML files under question would all have the same value. For example, suppose we want to import into database XML files that describe various metal parts, but the files themselves do not denote that they are metal. The database holds information for parts made of different materials (e.g. plastic, rubber, glass...etc). When importing the map file can simply map “metal” to the materials field in the database. `<value>` is the only datasource that can be empty.

**Example map file:**

```
<mapset>
2  <map>
     <dbfield>parts.materialType</dbfield>
4     <value>metal</value>
  </map>
6 </mapset>
```

The field `parts.materialType` is assigned the string “metal”. A row is inserted into the table `parts` with `metal` assigned to the field `materialType`.

### 2.3 <prompt>

Sometimes it may be necessary to fill a database field with a value that is not static across the import (via the `<value>` option) and a value that does not have mapping in the XML file. By specifying the prompt value, the import script stops and waits for an input before proceeding to the next mapping. For each prompt request, an optional default value can be used in case where nothing is supplied to the prompt. In the absence of a default value a non-empty value is required for the import operation to proceed. The default value supplied is any datasource that can be used in a map block (value, node, constant). Modifiers for the prompt are:

1. `<default>` - the default datasource to use
2. `<filter>` - the name of the PHP function (either user defined, or built-in) to call on the data before insertion into the database. If the filter function should return nothing, the prompt will forever ask for input.

#### Example map file:

```
<mapset>
2  <map>
    <dbfield>parts.materialType</dbfield>
4  <prompt/>
    </map>
6  </mapset>
```

Above map file prompts the user for the value to assign to the `materialType` of the `parts` table.

#### Example map file:

```
<mapset>
2  <map>
    <dbfield>parts.materialType</dbfield>
4  <prompt>
    <default>
6  <value>metal</value>
    <default />
8  </prompt>
    </map>
10 </mapset>
```

Prompts suggesting 'metal' to be the value. If the user enters nothing 'metal' will be used. Instead of using `<value>`, we could have used `<node>` to supply a default value that may exist in the import file.

### 2.4 <node>

Through the use of `xPath`, text nodes in the XML are selected and mapped to the associated database field. Multiple `<node>` definitions can be made to map to a single database field. The results of each evaluation will be concatenated

in the same order the XPath expression appear in the mapping file. An XPath expression can either evaluate to a single node or a collection of nodes, but all nodes must be text nodes. Due to PHP's implementation of XPath, the XPath expressions are limited to XPath1.0. To help get around some of those limitations additional optional modifiers can be applied to the result of each XPath expression:

1. <match> - a regular expression that must match the result of the evaluation. Useful for selecting a subset of nodes with the same path.
2. <empty> - use to specify that the node can resolve to an empty value, after applying regular expressions and the filter function.
3. <path> - the XPath expression to use. Only used when optional modifiers are used. A PHP function may be used as well to select certain nodes. (e.g. //book[php:functionString("substr", title, 0, 3) = "PHP"] ).
4. <filter> - the php function to call just before insertion into the database, but after the application of the <match> parameter. The function expects a string parameter and must return a string or nothing at all.
5. <nopath> - not applied to the result of the XPath expression, but applied when the XPath expression fails to resolve to an existing node, this modifier contains a list of alternative mappings to use in an attempt to save the map from failing. If all alternative mappings fail or any normal map fails, the XML import fails. Supplying a <value> datasource as the last alternative map will prevent failure. The trade off is usually an inaccurate import. Any existence of <dbField> in an alternative mapping will completely override the parent mapping. While it may be possible to construct a hierarchy of alternative mappings, it only a side effect and should not be relied upon.

### Example map file:

```

1 <mapset>
2   <map>
3     <dbfield>parts.name</dbfield>
4     <node>/part/partinfo/name</node>
5   </map>
6 </mapset>

```

Mapping with a node with only an XPath expression.

```

1 <mapset>
2   <map>
3     <dbfield>parts.name</dbfield>
4     <node>
5       <path>/part/partinfo/name</path>
6       <match>^10\..*$</match>
7     </node>
8   </map>
9 </mapset>

```

Mapping using an xPath expression to map from node(s) in the importing XML file matching the regular expression.

```

1 <mapset>
  <map>
3     <dbfield>parts.name</dbfield>
     <node>
5     <path>/part/partinfo/name</path>
     <filter>expandName</filter>
7     <node>
     </map>
9 </mapset>

```

Nodes resulting from the xPath query are each run through the filter function *expandName* before being concatenated together as a single value to map to *parts.name*. If a regular expression was supplied via *<match>*, only nodes from the xPath query results matching the regular expression are supplied to the filter function.

```

1 <mapset>
  <map>
3     <dbfield>parts.name</dbfield>
     <node>
5     <path>/part/partinfo/badnamenode</path>
     <filter>expandName</filter>
7     <nopath>
     <map>
9     <node>/part/partinfo/name1</node>
     </map>
11    <map>
     <node>
13    <path>/part/partinfo/name2</path>
     <filter>trim</filter>
15    </node>
     </map>
17    <map>
     <value>missing name</value>
19    </map>
     </nopath>
21    <node>
     </map>
23 </mapset>

```

With this example we handle the case of, */part/partinfo/badnamenode*, not existing or having errors. If */part/partinfo/badnamenode* does not return a result, the maps listing in *<nopath>* are evaluated in order until one is found that returns a result. If */part/partinfo/name1* returns nothing, then */part/partinfo/name2* is evaluated and passed through the built-in php function *trim*. After trimming the resulting node(s), the system checks to see if it has a non-empty value. If the value is empty, the the map failed and instead the string “missing name” is mapped to “parts.name”. To enable failed mapping to map to empty

string, specify the `<empty>`. Another method is to supply an alternative mapping with an empty `<value>`.

## 2.5 `<constant>`

The `<constant>` datasource is used as a means to use runtime constants. In particular this could be foreign keys, or surrogate keys via some `auto_generate` field property.

1. `{table}.LAST_INSERT_ID` - the last insert id use to insert a row in a table. The table reference must be one mentioned in some previous mapping.

```
1 <mapset>
  <map>
3   <dbfield>dataset.description</dbfield>
   <node>/metadata/idinfo/descript/abstract</node>
5 </map>
  <map>
7   <dbfield>dataset.discipline</dbfield>
   <prompt/>
9 </map>
  <map>
11  <dbfield>dataset_creator.creator_dataset_id</dbfield>
   <constant>dataset.LAST_INSERT_ID</constant>
13 </map>
  <map>
15  <dbfield>dataset_creator.creator</dbfield>
   <node>/metadata/idinfo/citation/citeinfo/origin</node>
17 </map>
</mapset>
```

The dataset table has a autoincrement field and so when a row is inserted, the ID is represented in the constants `dataset.LAST_INSERT_ID`, which is then used in the creator table import to associated the creator of the dataset.

## 3 Limitations

XML database files with a high frequency of identical nodes, but different text content, cannot be imported because `xml2db` assumes each file is its own record. The problem exists with using `xPath` queries to specify which text nodes to pull into the database. One way around this is to explicitly define the mapping for each record via its index in the file (e.g. `/path/to/node[1]`, `/path/to/node[2]`...etc), but making the mapping file quite large in the process. This method is impractical for large numbers of records. Future versions will lift this limitation. Also, it is unable to operate on more than one file at a time.