PilotFish eiConsole
SQLXML User Guide

2017 Edition

Table of Contents

1. Introduction

SQLXML, or SQL in XML, is a database-generic approach to expressing SQL statements in the PilotFish application. Its purpose is to allow complex SQL operations to be scripted dynamically and populated with content from inbound data during a PilotFish transaction. Effectively, the inbound data is mapped onto a series of SQL statements, represented by the XML structure.

It provides a three-level abstraction layer for working with the database it will be executed against. The levels define specific types of operations: **Generic Statements**, **Complex Statements**, and **General Workflow**.

2. Use With XSLT

In most cases, SQLXML will be generated using XSLT in the PilotFish Data Mapper. In many ways, SQLXML is very similar in concept to XSLT: a series of special XML elements that provide instructions to a processor that evaluates them. Because of how XSLT can dynamically build an output format based on the values in an input format, it is ideal to use the Data Mapper to generate SQLXML using the data coming in.

However, it is important to always be aware that SQLXML and XSLT are completely independent of each other. They are each evaluated separately, by totally separate components. The XSLT created by the Data Mapper is evaluated during the XSLT Transformation stage, whereas the SQLXML is evaluated by a subsequent stage, usually the PilotFish Database SQL Transport.

This means that any XSLT instruction elements will only be used for creating the SQLXML document, and will not be present when the SQLXML is being evaluated. A good example would be iteration. An <xsl:for-each /> element will be used by the XSLT Transformation to produce XML output during its iteration. A <sql:Iterate /> (covered later in this document) element will achieve the same iteration, but it will run only later when the SQLXML is evaluated.

The simple rule of thumb is that anything with the XSLT namespace prefix will only be present for the XSLT Transformation, while anything with the SQLXML namespace prefix will only be evaluated later on.

3. Quick Overview

This is a quick overview of what a SQLXML document looks like and the basic features of how it works. All of these elements will be explored in more detail in later sections in this document.

```
<!-- (A) Declare the SQLXML root tag along with the SQLXML namespace -->
<sql:SQLXML xmlns:sql="http://pilotfish.sqlxml">
        <!-- (B) Use a convenience wrapper around a SELECT statement -->
        <sql:Select into="resultset"> <!-- (C) Store SELECT result in a variable -->
                <PEOPLE>
                        <PERSON_ID />
                        <FIRST_NAME />
                        <LAST_NAME />
                        <GENDER />
                        <AGE />
                        <MEMBER key="true">1</MEMBER>
                </PEOPLE>
        </sql:Select>
        <!-- (D) Iterate over each record in a SQL ResultSet -->
        <sql:Iterate over="resultset" as="result">
                <!-- (E) Use a convenience wrapper around an INSERT statement -->
                <sql:Insert>
                        <MEMBERS>
                                <!-- (F) Use OGNL to retrieve values from the result to insert -->
                                <FIRST_NAME>ognl:#result.getFieldValue('FIRST_NAME')</FIRST_NAME>
                                <LAST_NAME>ognl:#result.getFieldValue('LAST_NAME')</LAST_NAME>
                                <GENDER>ognl:#result.getFieldValue('GENDER')</GENDER>
                                <AGE>ognl:#result.getFieldValue('AGE')</AGE>
                                <PERSON_ID>ognl:#result.getFieldValue('PERSON_ID')</PERSON_ID>
                        </MEMBERS>
                </sql:Insert>
        </sql:Iterate>
        <!-- (G) Execute a literal SQL statement -->
        <sql:Execute>
                <sql:SQL>UPDATE report SET last_update = '2017-01-01'</sql:SQL>
        </sql:Execute>
        <!-- (H) Output the results of the first query -->
        <sql:XMLOut var=”resultset” />
</sql:SQLXML>
```

(A): Every SQLXML document must start out with the standard root element and namespace declaration.

(B)/(E): SQLXML provides many convenience wrappers around common SQL statements, building them with simple XML rather than having to write the SQL directly.

(C): Variables are used to store results and other values so they can be manipulated later on.

(D): Flow control structures such as iteration are provided to allow more complex instructions on how to handle the various SQL operations.

(F): Accessing variables is done using a dynamic expression language, OGNL.

(G): If needed, literal SQL statements can be executed directly as well.

(H): Any ResultSet can be outputted in XML format so it can be manipulated by other stages in a PilotFish Route.

4. Declaration

To use SQLXML, a root element must be declared. This element must also have the SQLXML namespace assigned to it:

```
<sql:SQLXML xmlns:sql="http://pilotfish.sqlxml">
        <--Content -->
</sql:SQLXML>
```

5. SQLXML Settings

Some of SQLXML's internal behavior can be configured by setting attributes on the root (SQLXML) element. These are useful for controlling naming conventions of variables, escaping or qualifying identifiers like table or column names, and setting default values. Each of these is described briefly below:

- **identifierQuoteString** - Specifies the left-hand character used for wrapping identifiers like column or table names. Some databases allow for column names with whitespace or unusual characters which are wrapped in quotes or brackets.
- **identifierEndQuoteString** - Specifies the right-hand character (if different) used for wrapping identifiers. If not present, identifierQuoteString will be used.
- **tagNameChars** - Specifies which non-alphanumeric (A-Z, 0-9) characters are allowed in variable or tag names. Adding "_" will allow underscores in names, which is not allowed by default.
- **defaultNullString** - Specifies the default value for string-type columns
- **defaultNullNumber** - Specifies the default value for numeric-type columns
- **defaultNullDate** - Specifies the default value for date-type columns
- **identityQuery** - Defines a query used for determining primary keys for auto-incremented or triggered tables. Please see the "Identities in Inserts" section. The default value is for SQLServer.

An example of a SQLXML tag configured to allow underscores in variable and column names:

```
<sql:SQLXML xmlns:sql="http://pilotfish.sqlxml" tagNameChars="_">
        <--Content -->
</sql:SQLXML>
```

6. Variables

In any procedural set of instructions, variables are a crucial part of the workflow. SQLXML has broad variable support, allowing everything from a simple value to whole sets of records from the database to be stored in and accessed from a variable.

6.1. Simple Values

Simple values can be any kind of value set in a variable using the **Assign** element. These tend to be static items that are needed in other expressions later on. More details on using **Assign** can be found later on in this document.

6.2. SQL Results

A single record or an entire set of records from a SQL statement that retrieves data, such as a **Select**, can be stored in a variable. All SQLXML elements that perform SQL operations that can retrieve data include the ability to store that data into a variable, and this documentation will point out that capability for each one when it gets covered in later sections.

6.3. Accessing Variables

Variables are accessed using **Dynamic Expressions**, which are supported in most locations throughout SQLXML. A detailed description of how to work with **Dynamic Expressions** can be found in later on in this document.

6.4. Parent and Child Variables

Variables in SQLXML can have a parent-child style relationship to one another. Any one variable can be set as the parent of any number of child variables. The value of doing this mostly applies to working with the **XMLOut** element, which produces the output of a SQLXML operation and is covered in more detail in a later section. When the contents of a variable are outputted, all its child variables are outputted along with it.

Defining a parent-child relationship between variables is extremely simple, and is done using dot-notation. The parent variable, for obvious reasons, must be created on its own first. After that, any variable that should be its child is named using the following convention: “parentName.childName”. The dot indicates that “childName” is a child variable of “parentName”, and it is stored in this manner.

Here is an example of building this parent/child hierarchy.

```
<sql:Select into=”parentVariable”>
        <!-- Do Select -->
</sql:Select>
<sql:Select into=”parentVariable.childVariable”>
        <!-- Do Select -->
</sql:Select>
<!-- Output parent and child here -->
<sql:XMLOut var=”parentVariable” />
```

7. Dynamic Expressions

Dynamic expressions are possible in many locations throughout the SQLXML structure. The main value of using these expressions is working with variables and SQL ResultSets, although they also provide more complex capabilities. Generally, any place that accepts a value can accept an **OGNL** expression. This expression must be prefixed by “ognl:”, so that the parser treats it properly.

In specific places in some of the **General Workflow** structures, the expression can be provided without the “ognl:” prefix. These places are specifically intended to expect an **OGNL** expression to be the only thing passed to them. The documentation here will specify when this is so for the specific places that is true.

In addition, all variables, including SQL ResultSets, can only be accessed using **OGNL**. Variables in **OGNL** are referenced using the name they are assigned to, plus a “#” sign. For example: “ognl:#variableName”.

Please note that certain operators in these expressions need to be escaped because this format is an XML structure. For example, the less-than operator “<” becomes “<”. Please consult online references for the escaped versions of other operators that are XML control characters.

8. SQLXML Component Overview

This is a quick overview of the categories of components that are a part of the SQLXML structure.

8.1. Generic Statements

Generic Statements are a database-agnostic way SQLXML provides to perform basic SQL operations. Things like **Insert**, **InsertOrUpdate**, **Select**, **SelectOrInsertAndSelect**, **Update**, and **Delete** are examples of the kinds of operations covered here. Using the simple XML structures provided by SQLXML, these kind of queries can be built without writing any actual SQL, thus allowing the provided JDBC driver and the SQLXML parser to build the actual statements themselves. The advantage to this is that these structures can be ported between different database engines with virtually no changes.

8.2. Complex Statements

For more advanced SQL operations, SQLXML supports **Complex Statement**s. These are covered by items like **CallProc** and **Execute**. These elements allow raw SQL statements to be provided, so that complex operations can be executed against the database. All of these statements can

be safely parameterized with dynamic values, thus allowing inbound data to populate the query while protecting against SQL Injection.

8.3. General Workflow

General Workflow elements in SQLXML include things like **Assign**, **Iterate**, **If**, **While**, **TryCatch**, and **XMLOut**. These are business logic constructs, which control the general flow of operations as the various queries execute. They also perform other support functions, like error handling and creating variables. Finally, they handle outputting data at the end of the operation, so that it can be handled by other parts of the PilotFish workflow.

9. Generic Statements

This section will break down the elements that make up **Generic Statements** within SQLXML.

9.1. Select and SelectOrInsertAndSelect

The **Select** structure represents the basic SQL "select" query, used to retrieve records from the database. The **SelectOrInsertAndSelect** structure is a convenience wrapper around a common SQL operation. It is structured identically to the **Select**, however it has the added behavior of performing an insert operation if no records are returned by the select, and then performing another select to return the just-inserted record.

This is what a **Select** tends to look like:

```
<sql:Select into="RecordSetName">
        <tablename>
                <columnname key="true">value</columnname>
                <columnname />
                <columnname />
        </tablename>
</sql:Select>
```

The "tablename" element is the name of the table in the database that data should be selected from.

The "columnname" elements are the names of columns for whom data should be retrieved. In regular SQL, it would appear this way: "select columnname, columnname, columnname…".

The "key" attribute one one of the "columnname" elements tells the parser to treat that like part of a WHERE clause. The column in question and its value will be used like the following SQL: "…where columname = value". This is the only type of WHERE clause supported here, for more complex WHERE clauses see the **Execute** element under the Complex Statements section.

The attribute "into", on the parent "Select" element, sets a variable name that the results will be stored in. This variable name is how the results can be accessed elsewhere in the SQLXML document.

For the **SelectOrInsertAndSelect**, default values can be assigned for all columnnames elements. Only values for elements with the "key=true" attribute will be used for a where clause, and if no matching records are found, all values will be inserted into the database.

9.2. Insert, Update, Delete, and InsertOrUpdate

These structures are all used to modify content within the database. **Insert**, **Update**, and **Delete** are just the based CRUD operations. **InsertOrUpdate** is a convenience wrapper that updates a record if a match exists, and inserts it otherwise.

All of these structures tend to look very similar, except with a different parent command element:

```
<sql:Insert>
        <tablename>
                <columnname key="true">value</columnname>
                <columnname key="true">value</columnname>
        </tablename>
</sql:Insert>
```

The first element, “Insert”, is the parent command element for the operation. It matches the type of operation being done, either **Insert**, **Update**, **Delete**, or **InsertOrUpdate**.

The “tablename” element is the name of the table in the database that is being manipulated.

The “columnname” elements are the names of the columns where data is being manipulated. For **Insert**, **Update**, or **InsertOrUpdate**, they are the specific columns that data is being manipulated in. Values assigned to these elements get populated in those columns. For **Delete** operations, they are used simply as keys to denote which records are being deleted, and are treated like a where clause.

The “key” attribute specifies that a given column value should be used in a where clause. For **Insert**, this has no value. For **Update**, **InsertOrUpdate**, or **Delete**, this determines which matching records get affected by the SQL operation. Columns marked with “key=true” are populated in a SQL where clause like this: “…where columnname = value”.

9.3. Identities in Inserts

Database tables are often configured so that their primary key is automatically set for a given row when it is first inserted, or later based on some trigger or procedure. In these cases, you'll often need to query the database to determine the value of the created identifier for the last inserted row. SQLXML has a special "identity" attribute to make this easier for basic inserts.

By adding a "identity" attribute with a desired result variable name to the "Insert" tag, SQLXML will use the "identityQuery" SQLXML attribute (see "SQLXML Settings") to execute a query against the target table and store the result into the named variable. Here's an example showing a person being inserted, then an address record that references the person (via FK_PEOPLE) making use of the value:

```
<SQLXML xmlns="http://pilotfish.sqlxml" identityQuery="SELECT LAST_INSERT_ID()">
	<Insert identity="person_id">
		<PEOPLE>
			<FIRST_NAME>Luke</FIRST_NAME>
			<LAST_NAME>Kirk</LAST_NAME>
		</PEOPLE>
	</Insert>
	<Insert>
		<ADDRESS>
			<FK_PEOPLE>ognl:#person_id</FK_PEOPLE>
			<LINE_1>123 Space Ln</LINE_1>
		</ADDRESS>
	</Insert>
</SQLXML>
```

10. Complex Statements

This section will break down the elements that make up **Complex Statements** within SQLXML.

10.1. Execute

The **Execute** structure is used for executing complex SQL queries. It wraps around a plain text SQL statement of any level of complexity, and simply executes it against the database. Because this is raw SQL, there is nothing database-agnostic about it.

The **Execute** structure looks like this:

```
<sql:Execute into="secondVariableName" as="recordName">
	<sql:SQL>
		SELECT
			someValues
		FROM
			TableNameQualifier.TableName
		WHERE
			someCondition = ? AND
			someOtherCondition = ?
	</sql:SQL>
	<sql:Params>Condition1</sql:Params>
	<sql:Params>ognl:#existingVariableValue</sql:Params>
</sql:Execute>
```

The “SQL” child element is what wraps around the raw SQL statement. Note how it is a parameterized statement, with “?” representing places where parameters can be provided. This mirrors the API for constructs like Java’s PreparedStatement, which are used to protect against SQL Injection.

The “Params” elements represent SQL parameters. For each “?” in the raw query, there must be a matching “Params” element with a value. The values of the “Params” elements will be placed into the query sequentially, so the order of these elements is important.

Execute has the attributes “into” and “as”. If the query returns data, these attributes determine how that data returned will be handled. “into” defines the name of the **RecordSet**, the collection of multiple records that are returned. “as” defines the name of an individual record within the **RecordSet**.

10.2. CallProc

The **CallProc** structure handles calling a stored procedure in the database.

The **CallProc** structure looks like this:

```
<sql:CallProc into="someVariableName">
        <sql:SQL>call storedProcedureName(?,?,?)</sql:SQL>
        < sql:Param mode="IN" name="parameter1Name" type="12">someValue</ sql:Param>
        < sql:Param mode="IN" name="parameter2Name" type="12">someValue</ sql:Param>
        < sql:Param mode="OUT" name="parameter3Name" type="12" />
</sql:CallProc>
```

The “SQL” child element is where the SQL code to call the procedure is written. This code is parameterized just like **Execute** structures, however these parameters need to conform to procedure standards.

The “Param” child elements are the parameters for the procedure. Unlike with **Execute**, procedures require additional metadata for these parameters, which are specified in attributes. “mode” specifies if it’s an “IN”, “OUT”, or “INOUT” parameter. “name” is the name of the parameter. And “type” is the numeric data type code. Please consult JDBC documentation for the type codes.

The text values of “Param” elements are what actually gets passed in as the parameter value. “OUT” parameters shouldn’t have a value, since it is populated by the results of the procedure.

Like **Select**, **CallProc** has the attribute “into”. If the query returns data, this attribute determines how that data returned will be handled. “into” defines the name of the **RecordSet**, the collection of multiple records that are returned.

11. General Workflow

This section will break down the elements that make up **General Workflow** within SQLXML.

11.1. Assign

The **Assign** structure assigns a value from a dynamic **OGNL** expression to a variable. It looks like this:

```
<sql:Assign exp="#variable.substring(0,5)" name="substring" />
```

The “name” attribute is the name of the variable that can be used to access the value later.

The “exp” attribute is the **OGNL** expression that is used for the value. Because this attribute always expects to receive an **OGNL** expression, the prefix (“ognl:”) is not required here.

11.2. If

The **If** structure is used to provide flow control in the form of conditional logic. In many cases, complex SQLXML operations will need to operate conditionally based on data returned from the database. This structure uses a dynamic expression to generate a Boolean value, which determines whether or not the structures within it execute.

The **If** structure looks like this:

```
<sql:If test="#itemName.getFieldValue('COLUMNNAME') &lt; 1">
        <!-- Operations -->
</sql:If>
```

The “test” attribute is where the dynamic expression to be evaluated is placed. This expression will always be in **OGNL**, so the prefix (“ognl:”) is not required here. This expression is expected to evaluate to produce a Boolean value. If that value is true, then the child operations will execute, otherwise they will not.

More details about how to access previous SQL ResultSets are provided in the ResultSet Variables section.

11.3. Iterate

The **Iterate** structure is used to provide flow control in the form of iteration. In many cases, complex SQLXML operations will need to iterate over a set of results returned from the database, and perform operations with each record. This structure performs that task.

The **Iterate** structure looks like this:

```
<sql:Iterate over="policies" as="policy">
        <!-- Operations -->
</sql:Iterate>
```

The “over” attribute is the variable that is being iterated over. **Iterate** is meant to loop over all the records in a **RecordSet**, so this variable must be the name of a **RecordSet** variable returned by a **Select**, **Execute**, or other structure that returns such a value.

The “as” attribute is the variable that represents an individual record. This is how a single record from the **RecordSet** is accessed during the iteration. The variable named by “as” is always the record for the current round of the iteration.

11.4. While

The **While** structure is another form of looping flow control. Unlike **Iterate**, **While** isn’t intended to simply loop over all the records in a **ResultSet**. It accepts a dynamic **OGNL** expression, and as long as that expression evaluates to true, it will keep executing.

The **While** structure looks like this:

```
<sql:While test="#count &lt; 5">
        <!-- Operations -->
        <sql:Assign exp="#count++ " name="count" />
</ sql:While>
```

The “test” attribute is used to determine whether or not the loop keeps running. It always expects to receive an **OGNL** expression, so the prefix (“ognl:”) is not required here. It must evaluate to a Boolean expression, and as long as it is true the loop will keep running.

Any SQLXML structure can be placed within the loop. However, the example includes an **Assign** structure. This is because the loop must alter the condition it is testing in some way to ensure that it ultimately ends. Using a variable, and modifying that variable with an **OGNL** expression within the loop, is the simplest way to accomplish this.

11.5. TryCatch

The **TryCatch** structure provides special error handling. When working with databases, SQL errors can be common. **TryCatch** provides the common try-catch-finally syntax for handling errors in special ways.

The **TryCatch** structure looks like this:

```
< sql:TryCatch>
        < sql:Try>
                <!-- Operations -->
        </ sql:Try>
        < sql:Catch>
                <!-- Operations -->
        </ sql:Catch>
        < sql:Finally>
                <!-- Operations -->
        </ sql:Finally>
</sql:TryCatch>
```

The **Try** element wraps around logic that could potentially throw an error. Any SQLXML structure can be placed within it.

The **Catch** element wraps around error handling logic. Any SQLXML structure can be placed within it, however this will only execute if an error occurs within the “Try” section.

Also, within the **Catch** element, there is an implicit variable named “EXCEPTION”. This contains the Java Exception object that was thrown, and can be accessed using OGNL expressions.

The **Finally** element is the post-process structure. Any SQLXML structure can be placed within it, and it will be executed after the **Try** and (possibly) **Catch** sections have finished executing.

11.6. XMLOut

The **XMLOut** structure outputs values to an XML structure produced after the SQLXML has finished being executed. The basic **XMLOut** structure looks like this:

```
<sql:XMLOut var="ResultSet" />
```

The “var” attribute is where the name of a variable is provided. This is where the name of the **ResultSet** variable is provided. All of the records of that **ResultSet** will be outputted.

This type of **XMLOut** can only be used once, and can be thought of as the “return” statement in most programming functions. However, there are times when data will have to be outputted at multiple points during the SQLXML execution. In this case, a separate type of **XMLOut** is used, with different attributes:

```
<sql:XMLOut appendTo="ResultSet" var="AdditionalData" />
```

This type of **XMLOut** can only be used after the earlier one has already been declared. The “appendTo”attribute specifies the name of a **ResultSet** variable that has already been passed into the **XMLOut**, and should be appended to. “var”, in this case, is the new **ResultSet** variable whose contents should be appended to the existing one.

Because “var” and “appendTo” always expect to receive **ResultSet** variable names, they are implicitly **OGNL**, and do not need either the prefix (“ognl:”) or the variable marker (“#”). This only expects that specific variable name, and cannot be provided a more complex expression.

Here is an example for an **XMLOut** for a variable named “Contracts”:

```
<EIPData>
	<CONTRACTS>
		<RECORD>
			<CONTRACT_ID>1</CONTRACT_ID>
			<CONTRACT_NAME>First Contract</CONTRACT_NAME>
			<CONTRACT_DATE>1/1/2017</CONTRACT_DATE>
		</RECORD>
		<RECORD>
			<<CONTRACT_ID>2</CONTRACT_ID>
			<CONTRACT_NAME>Second Contract</CONTRACT_NAME>
			<CONTRACT_DATE>1/2/2017</CONTRACT_DATE>
		</RECORD>
		<RECORD>
			<CONTRACT_ID>3</CONTRACT_ID>
			<CONTRACT_NAME>Third Contract</CONTRACT_NAME>
			<CONTRACT_DATE>1/3/2017</CONTRACT_DATE>
		</RECORD>
	</CONTRACTS>
</EIPData>
```

12. ResultSet Variables

PilotFish's internal handling of SQL **ResultSets** has a very specific API. Because **ResultSets** can only be manipulated by **OGNL** expressions, this API is very important when working with SQLXML. The following are the API structures of the Java classes used internally when SQLXML is processed.

12.1. RecordSetVariable

This variable wraps around a collection of records returned by a database operation. It has one public method:

RecordVariable[] getRecords();

This method returns a Java Array of all of the records that it contains. Each record is of the type **RecordVariable**. All operations that are valid for accessing a Java Array can be used on the return values. Always be aware that the **RecordSetVariable** could possibly be empty.

To access this method in SQLXML using OGNL, the following expression is used:

ognl:#firstVariableName.getRecords()

12.2. RecordVariable

This variable wraps around a single record from the database. The fields within it are accessed by using their respective column names. It has one public method:

String getFieldValue(String columnName);

This method returns a String (text) value from the underlying record. It uses the name of that field's column to identify and retrieve it. Please note that for more complex SQL queries, faux column names can be specified using the "AS" statement in the SQL query. Please consult references for SQL syntax for more information.

To access this method in SQLXML using OGNL, the following expression is used:

ognl:#secondVariableName.getFieldValue('columnName')

12.3. RecordSetVariable and RecodVariable Uses

When SQLXML populates a variable with results, it will dynamically choose whether or not to use a **RecordSetVariable** or a **RecordVariable**. If there are more than one record, a **RecordSetVariable** will be used. If there is only one record, a **RecordVariable** will be used. Keep this in mind when writing SQLXML handling logic.

12.4. Easy Handling Using Iteration

The easiest way to handle the results returned in SQLXML is using the Iterate element. As was covered earlier in the section on this element, **Iterate** will take the result and go over it one row at a time, returning each one individually. Whether a single row or multiple are returned, **Iterate** handles it perfectly.

Iterate also provides consistency when knowing which OGNL expressions to use. The object **Iterate** returns to access is always an individual **RecordVariable**, making it much easier to work with the result.

13. Handling Transactions

By default, SQLXML is evaluated in a transactional way. This means that individual SQL statements are not committed as they are evaluated, and instead the entire transaction is committed in its entirety upon the conclusion of the entire SQLXML evaluation.

PilotFish modules that evaluate and execute SQLXML, such as the Database SQL Transport, include an option to turn autocommit on. If chosen, each SQL statement will be committed individually as they are evaluated.

If it is needed to perform an explicit commit in mid-document, this can be easily done using an **Execute** element:

```
<sql:Execute>
        <sql:SQL>COMMIT</sql:SQL>
</sql:Execute>
```

14. Complete Examples

The following are several complete examples of the SQLXML concepts and structures that have been described here.

14.1. Multiple Inserts With Try-Catch

In this example, multiple SQL Inserts are done, each one wrapped in a Try-Catch block. If an error occurs, a record is inserted into an error table recording the cause of the error.

```
<?xml version="1.0" encoding="UTF-8"?>
<sql:SQLXML xmlns:sql="http://pilotfish.sqlxml">
	<sql:TryCatch>
		<sql:Try>
			<sql:Insert>
				<PEOPLE>
					<PERSON_ID>1</PERSON_ID>
					<FIRST_NAME>Bob</FIRST_NAME>
					<LAST_NAME>Saget</LAST_NAME>
					<AGE>60</AGE>
				</PEOPLE>
			</sql:Insert>
		</sql:Try>
		<sql:Catch>
			<sql:Insert>
				<ERRORS>
					<DATE>05/03/2017</DATE>
					<TABLE>PEOPLE</TABLE>
					<ERROR>ognl:#EXCEPTION.getMessage()</ERROR>
				</ERRORS>
			</sql:Insert>
		</sql:Catch>
	</sql:TryCatch>
	<sql:TryCatch>
		<sql:Try>
			<sql:Insert>
				<PEOPLE>
					<PERSON_ID>2</PERSON_ID>
					<FIRST_NAME>John</FIRST_NAME>
					<LAST_NAME>Doe</LAST_NAME>
					<AGE>22</AGE>
				</PEOPLE>
			</sql:Insert>
		</sql:Try>
		<sql:Catch>
			<sql:Insert>
				<ERRORS>
					<DATE>05/03/2017</DATE>
					<TABLE>PEOPLE</TABLE>
					<ERROR>ognl:#EXCEPTION.getMessage()</ERROR>
				</ERRORS>
			</sql:Insert>
		</sql:Catch>
	</sql:TryCatch>
</sql:SQLXML>
```

14.2. Select, Iteration with Multiple Inserts, and Output

This example performs a **Select** and iterates over the **ResultSet**. It does an insert into another table for each found record, retrieving a field value from it. Lastly, it outputs the ResultSet via **XMLOut**.

```
<?xml version="1.0" encoding="UTF-8"?>
<sql:SQLXML xmlns:sql="http://pilotfish.sqlxml">
	<sql:Select into="results">
		<PEOPLE>
			<PERSON_ID />
			<FIRST_NAME />
			<LAST_NAME />
			<AGE />
		</PEOPLE>
	</sql:Select>
	<sql:Iterate over="results" as="result">
		<sql:Insert>
			<MATCHES>

	<MATCHED_PERSON_ID>ognl:#result.getFieldValue('PERSON_ID')</MATCHED_PERSON_ID>
			</MATCHES>
		</sql:Insert>
	</sql:Iterate>
	<sql:XMLOut var="results" />
</sql:SQLXML>
```