

# System iNetwork

- [Subscribe](#)
- [My account](#)
- [Log out](#)
- [Contact Us](#)
- [Advertise](#)

Search

GO

- [Forums](#)
- [Archives](#)
- [Code](#)
- [Blogs](#)
- [Podcasts](#)
- [Webcasts](#)
- [e-Learning](#)
- [Guides](#)
- [Newsletters](#)
- [About Us](#)

- [Contact Us](#)
- [About the Network](#)
- [Tech Editor Profiles](#)
- [Editorial Calendar](#)
- [Writers Kit](#)
- [Advertise](#)



- [RPG Programming](#)
- [Other Languages](#)
- [Application Development](#)
- [Database/SQL](#)
- [Availability](#)
- [Security](#)
- [Systems Management](#)

- [Networking](#)

- [IT Mgmt/Careers](#)

- 

- 

- 

- 

- 

- 

[Home](#) » [Content](#)

## K.I.S.S. XML

Article ID: 62565

Posted January 1st, 2009

in

- [Other Languages](#)

Generate XML documents from DB2 by keeping it short and simple

By:

[David Andruchuk](#)

According to Wikipedia, the K.I.S.S. principle has many meanings, but the one I prefer to use is Keep It Short and Simple. This explanation best states that design simplicity should be a key goal and that unnecessary complexity should be avoided. K.I.S.S. serves as a useful principle in a wide array of disciplines such as software development, animation, engineering, and strategic planning. Building an XML document from System i database tables is no exception.

The ability to transform database records into XML documents is increasingly important for data interchange over the Internet, particularly when using XML-based web services. Many platforms already support databases that allow the use of SQL to generate an XML document natively, but as of release 6.1, DB2 for i is not one of them. DB2 for i does not have native XML or pure XML support yet. An IBM i middleware product, DB2 XML Extender, does support transforming DB2 queries into XML, but it isn't free, can be complex to configure, and can't always be easily integrated into existing applications. Fortunately, there is a short and simple solution to the problem: creating XML documents using an SQL stored procedure, a technique that handles both simple and somewhat complex XML document composition on the System i. But before we delve into learning how to generate XML documents from DB2 using this approach, let's quickly review stored procedures.

## Stored Procedure Refresher

A stored procedure is a program that can be called to perform operations and can include both host language statements and SQL statements. Procedures in SQL provide the same benefits as procedures in a host language. An SQL stored procedure is one in which the body of the program is written in the SQL Persistent Stored Module (SQL PSM) language. SQL PSM is extremely powerful and provides much of the functionality found in traditional languages such as C. (For full details on SQL PSM, see Chapter 3, "Introduction to the SQL Persistent Stored Module in DB2 UDB for iSeries," in *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries* (SG24-6503) at <http://www.redbooks.ibm.com/>.)

You define a procedure as either an SQL procedure or an external procedure. An external procedure can be any supported HLL program (except System/36 programs and procedures) or a REXX procedure. An SQL procedure is defined entirely in SQL and can contain SQL statements that include SQL control statements.

## Stepping Through an Example

A working example program illustrates this technique and serves as a useful template for your own XML SQL stored procedure. The example uses a pure SQL procedure to perform our XML composition, since our K.I.S.S. principle implies avoiding external HLL programs if possible.

You can choose the development environment to use to code your SQL Procedure Language (SQL PL) source. IBM Rational Application Developer, WDSi, IBM Navigator for i, Notepad or other text editors on your workstation, or PDM on the System i can all provide the correct application development platform environment to build our SQL stored procedure.

The example SQL procedure merges two tables, Employee Master and Department Master, to yield an XML document containing a subset of the columns from both. The resulting XML document contains the Employee Number, First and Last Names, Work Department, and Salary columns from Employee Master ([Figure 1](#)) and Department Name and Location columns from Department Master ([Figure 2](#)). Because the data requested resides in two tables, our SQL procedure will have to join the tables to retrieve the appropriate department information from the Department Master table (more on how to do this in a moment).

## Writing the Code

So now with our requirements defined, we can begin writing the SQL PL code ([Figure 3](#)). We define an SQL procedure to the system using the CREATE PROCEDURE statement, which adds procedure and parameter definitions to the system catalog tables SYSROUTINES and SYSPARMS. These definitions are then accessible by any SQL CALL statement on the system, including but not limited to STRSQL, Run SQL statement in Navigator for i, or any HLL (e.g., RPG, Java, C) SQL CALL statement.

Let's begin investigating the SQL PL code for a clearer understanding of what is being accomplished in our short and simple design. The DROP PROCEDURE statement at the start of the SQL PL source member guarantees that the existing procedure will be dropped from the database before creating a new procedure to generate our XML document. That way, we have the latest version registered in the system.

The CREATE PROCEDURE statement then defines a new procedure named GENEMDPXML (Generate Employee Department XML) with one argument. This simple example does not return any values to the caller because we want to create the XML document only for a one-time report. The input argument specifies the XML format for the resulting XML document — TREE or STRING. The TREE format encodes each XML element on a separate line; the STRING format keeps related elements on a single line.

After specifying that our procedure language is SQL and defining the beginning of the procedure code block, we declare the variable to contain our working XML script: a character string of 2,000 bytes. To store the generated XML code, we create a temporary table in library QTEMP, named the same as the procedure (GENEMDPXML), with one character field consisting of 2,000 bytes to match the string we just defined that contains our working XML script.

The XML document must have a header XML statement and one root element defined in it. The next two lines show how easy it is to insert the XML header and root element using the SQL INSERT statement's values parameter.

With the beginning of our XML document defined, we can now begin importing our Employee and Department Master data and formatting it as XML output. Using the SQL FOR statement, which retrieves statements for each row in a table, we specify an SQL variable name (Each\_EmpDept) and the cursor (EmpDept) that will retrieve all Employee and Department Master rows to build our XML document. (For full details about the SQL FOR statement, see Chapter 6, "SQL Control Statements," in *DB2 Universal Database for iSeries SQL Reference*.)

The SELECT statement defines the columns to retrieve from our two tables and the relationship between the two tables. In this case, specifying WORKDEPT in our WHERE clause will link the Department Master to the Employee Master. So each Employee row returned will have the related Department Master row retrieved by using the common key specified. The returned results will be in Employee Number order. Now that we've built our repeating FOR loop that uses the EmpDept cursor to retrieve the data, we can define what we want to *do* once the data is retrieved.

Just as the FOR statement creates the boundaries of the cursor to execute for all rows returned from our tables, the DO statement groups the processes that you wish to perform for each of those returned rows. This is where we define SQL procedure statements to build our XML document for our retrieved data. As you recall, the root XML element is <Employees>, which sets the top of our hierarchical structure of our XML document tree for all employees. For each

employee, we insert the <Employee> parent element group to separate each employee data block.

As part of generating our XML document, we check each data column retrieved from our database tables for the existence of predefined entities that will cause problems for our XML document. These character entities must be replaced with a predefined replacement text. That way, when these characters are processed in the XML document, or an HTML page, the appropriate substitution occurs. (For more details about XML and HTML character entity references, see the Wikipedia page "List of XML and HTML character entity references" at [tinyurl.com/ffl5g](http://tinyurl.com/ffl5g).) In our SQL PL, we perform this check on the Department Name column from the Department Master table only, but any column that is a name, description, or so on, should be checked as well to ensure that the data does not contain these special entity character values.

Next, we check the passed XMLFORMAT parameter for either the TREE or other value. To determine the XML document format, we use the IF statement. When the value is TREE, we insert each column as a row into our temporary work table wrapped by the applicable column's element name. For a value other than TREE, we concatenate the columns and element names in the 2,000-character string before inserting the columns into our temporary work table. Each data column is then preceded by the column name element literal (e.g., <Empno>), then trimmed to remove trailing blanks using the RTRIM scalar function trim operation (RTRIM(empno)), and concluded with the column name closing element literal (e.g., </Empno>).

Our final instruction is to close our parent element so that the employee data is closed out before the next employee block is processed. The ENDFOR statement specifies the end of the DO processing for the cursor definition previously set. We can then insert the end root element(</Employees>) to complete our XML document generation.

All that remains is to copy the generated XML data to a stream XML document in the IFS. To accomplish this task, we can use the CPYTOIMPF command as an argument to the system command QCMDEXC in the SQL PL, in a CLLE program, or even on the command line. Now, specify the CPYTOIMPF parameters and values within the QCMDEXC parameter string along with the total length of the command script (164.00000), prepare the dynamic SQL string, and then execute it.

A final housekeeping task remains: dropping our temporary work table created in QTEMP that held our XML document data. As you remember, we created this table during the execution of each call to the GENEMDPXML procedure, so if the table already exists, our procedure will error out.

## **Compiling the SQL Procedure**

With the code complete, we can now compile the SQL PL to create the procedure and register it on the system where it will be executed. As stated earlier, we define an SQL procedure to the system by using the CREATE PROCEDURE statement. We coded this statement in our SQL

PL source member, which then adds procedure and parameter definitions to the system catalog tables SYSROUTINES and SYSPARMS.

Any SQL CALL statement on the system can then access these definitions.

We have two options for our create process: use RUNSQLSTM on the command line or use Navigator for i. On your workstation command line, key the RUNSQLSTM command and prompt F4 ([Figure 4](#)). The Source file, Library, and Source member are fairly standard. The Naming parameter is set to \*SQL because we coded our temporary work table as QTEMP.GENEMDPXML. Had we used the system naming convention QTEMP/GENEMDPXML, we could have left the value as the default \*SYS. Pressing F10 (Additional parameter) lets you specify the Severity level and Default collection parameters.

Next, we set the Severity level to 30. Doing so allows for bypassing the Severity level 20 messages that will occur the first time we execute CREATE PROCEDURE, as our SQL procedure will not yet exist when we try to drop it.

## Executing the SQL Procedure

Now that we've created our SQL procedure, we need to execute the procedure and test that it works. Note that we also choose how we want to execute our procedure. The created SQL procedure can be called using the SQL CALL statement from any SQL interface (i.e., STRSQL, Run SQL Scripts in Navigator for i, a procedure in a service program if one has been created for such a purpose).

Before we execute the procedure, the directory that will store the generated XML document must already exist. If you remember from our coding earlier, we specified tempxml doc as our directory name in the CPYTOIMPF statement. You can use the CRTDIR command if the directory does not already exist on the system where you will execute the script. We're now ready to test the script by invoking it directly. From an SQL command line, such as the STRSQL environment, type

```
call qqpl.genemdpxml ('TREE')
```

Upon command completion, you should get the message "CALL Statement complete." If you call the procedure with any other argument, the resulting XML document will be string rather than tree format. [Figure 5](#) and [Figure 6](#) show a typical output XML document in each of these formats.

## Short and Simple

By keeping it short and simple, we were able to quickly reformat our Employee and Department Master tables and create a self-described exchange document to share with our external source. With the examples reviewed in this article, you should have the basic foundation for the design

and coding of an SQL stored procedure to generate an XML document. Now you can begin generating your own XML documents from your DB2 tables.

*David Andruchuk is president of Computer Systems Design Associates, Inc. He began on an IBM S/32 and has progressed through the midrange platform offerings during his 25-year career.*

Bookmark/Search this post with:



[Email this page](#)

[Printer-friendly version](#)

### Post new comment

Your name: pwesemann@sunga...

Comment: \*

#### Input format



Filtered HTML

Web page addresses and e-mail addresses turn into links automatically.

Allowed HTML tags: <a> <em> <strong> <cite> <code> <ul> <ol> <li> <dl> <dt> <dd>

Lines and paragraphs break automatically.

## Full HTML

Web page addresses and e-mail addresses turn into links automatically.

You may use [\[block:module=delta\] tags](#) to display the contents of block *delta* for module *module*.

### [More information about formatting options](#)

Math Question: \* 3 + 17 =

Solve this simple math problem and enter the result. E.g. for 1+3, enter 4.

Preview comment

[Acceptable Use Policy](#)

### Related Links

[The Direction of System i Technology, Part 1](#)

[Hibernate Your JDBC](#)

[Handle Null Values in JDBC from RPG](#)

[JDBC from RPG](#)

[Retrieve an SQL Result Set with RPG](#)

ProVIP Sponsors

- 
- 
- 
- 
- 

### Featured Links

### Sponsored Links

- Home
  - [Subscribe Now](#)
  - [Advertise](#)
  - [Contact Us](#)
  - [Feedback](#)
  - [Terms of Use](#)
  - [Trademarks](#)
  - [Privacy Statement](#)
- © 2009 Penton Media, Inc.