Generating Configuration Files: The Directors Cut

Jon Finke – Rensselaer Polytechnic Institute

ABSTRACT

The generation of system configuration files and other documents directly from a database has proven to be a very powerful technique. However, there were some limitations to this approach. With the introduction of Oracle 8i, and more specifically, the addition of support for XML, we have been able to eliminate many of these limitations and take the file generation and maintenance to a new level.

Introduction

At LISA 2000, I presented a paper [6] on generating system configuration files and other documents directly from data stored in a relational database. This approach had many strengths, such as allowing version control of documents and files, selective regeneration based on need (only regenerate files that have changed), centralizing the file generation logic. The mechanics of writing actual files was handled by a simple, generic file generation program and all of the hard work was done in the database. This model has worked well for us, and has proven to be quite flexible and convenient for the deployment of new files.

With all the strengths of this approach, we also hit some limitations. We found that the extraction of the data from the database required one particular skill set, while formatting that data for presentation or eventual end use required a different, often unrelated, skill set. Simply put, the extraction of the data required database manipulation skills, i.e., the ability to program in PL/SQL.[1] Formatting that extracted data required different skills and knowledge. This latter set of skills and knowledge could be, e.g.,

- The format and structure of OS configuration (e.g., /etc/passwd and /etc/group);
- typesetting skills (e.g., LaTeX, HTML); or
- how "corporate branding" (e.g., logos, templates) might affect web pages.

A second issue came up with the generation of web pages. As the visibility of web documents was growing, so was the interest by the "Marketing and Media Relations" folks in the actual appearance of the web pages, down to the use of graphics and buttons and the look and feel of the web pages. To make matters worse, this look and feel was constantly changing and evolving, and it was a chore to keep the generated web pages looking like the rest of the site's pages. Some of this could be handled via cascading style sheets, and some other tools to allow the graphics designers to specify sections of "boiler plate" within documents, but the basic structure and layout was still dictated by the PL/SQL code. Any change in what data elements were displayed, or even the order of columns in a table still required intervention by the programmers and that pleased neither the programmers nor the web folks.

Clearly, we needed a way to separate the extraction of the data and it's reformatting. This would let the database programmers concentrate on extracting the data without getting bogged down with the final format issues. Folks who were concerned with format, such as web designers, marketing managers, etc., could concentrate on the formatting process. With the release of Oracle 8i, support was added for the processing and transformation of *eXtensible Markup Language* (XML) [2] documents. This provided a method of extracting the data from the database in a display neutral format, and then allowing the web designers to apply whatever transforms and formatting that was required. This meant that once the file extraction routine was written, the PL/SQL programmers were done and the web designers could take over and make formatting changes as their needs and schedules dictated. In addition, since the database could invoke the translations internally, we were able to keep all of the version control and process monitoring [8] features that we found useful in the first version, and use the same generic program to actually write the files.

XML – A Brief Primer

The "parent" of XML is Standard Generalized Markup Language (SGML). Another offspring of SGML is HTML, which many of us are already familiar with. One of the problems with HTML, is that it is not extensible, or perhaps worse, it does get extended by some browsers and servers and not by others. I am certain that many of us have visited web pages that don't work quite right on our browser of choice. XML was designed to be extensible, yet contained so that applications developed to process an XML document, can deal with any XML document. A good discussion of this is available in Chapter 6 of *The LaTeX Web*

[1] PL/SQL is a procedural extension to Oracle that allows you to write programs that are stored and executed in the Oracle database itself [11].

Companion [9], including why XML will not run into this problem, unlike HTML.

But XML is not limited to document preparation. Another area where XML is used is in business to business applications to allow businesses to exchange information in a consistent manner. Inherent in an XML document, is the Document Type Definition (DTD). The DTD (or schema) provides a set of rules that not only defines the element names and types, but also the relationship between data elements in the XML document. This provides a way to validate an XML document to ensure that it is correct. This schema is in fact another XML document. By sharing these XML schemas, different parties can independently develop XML documents and interfaces that will inter-operate.

Using XML to Encode OS Configuration Files

But XML has more uses than just document preparation or exchanging information between business. We have the need to generate /etc/passwd files or the equivalent from our database and make it available to different kinds of systems. Although we could generate the traditional ":" delimited form of the file and then use tools and scripts to reformat that if needed (like for our LDAP servers), we face a problem if we need to include more information about an entry. We can add more fields, but we then risk breaking those tools and scripts. We can instead encode our password entry information in XML.

The XML version of our password file consists of two parts. The first, seen in Figure 1 is the DTD which defines how the rest of the file will be formatted. This can be very useful to people who want to process this data and allows us to mechanically verify the format of the data. The second part, seen in Figure 2 contains the actual password data enclosed by <etc_passwd> tags. Each record (or line in the /etc/passwd file), is in turn enclosed with the <pw_entry> tags. Within each record are a number of elements. The first six should look pretty familiar, but there are some additional fields that may be useful to other applications. We will see some examples of how these are used later on.

XML and Translation

One of the facilities provided with many implementations of XML, is the ability to use an *eXtensible Stylesheet Language* (XSL) [1] template (or style sheet) to transform an XML document into some other format such as HTML, LaTeX or whatever is needed. The important point being that this enable us to separate the extraction of the data from the database, from the details of the final output format, and in fact, the same XML data can be translated by several different templates to generate different types of documents.

```
<?xml version = '1.0'?>
<!DOCTYPE etc_passwd [
<!ELEMENT etc_passwd (pw_entry)*>
<!ELEMENT pw_entry (username?, unixuid?,
    gid?, gecos?, homedir?, shell?,
    person_id?, user_type?, platform?)>
<!ATTLIST pw_entry platform
                   CDATA #REQUIRED>
<!ELEMENT username (#PCDATA)>
<!ELEMENT unixuid (#PCDATA)>
<!ELEMENT gid (#PCDATA)>
<!ELEMENT gecos (#PCDATA)>
<!ELEMENT homedir (#PCDATA)>
<!ELEMENT shell (#PCDATA)>
<!ELEMENT person_id (#PCDATA)>
<!ELEMENT user_type (#PCDATA)>
<!ELEMENT platform (#PCDATA)>
]>
```

Figure 1: XML DTD for /etc/passwd.

In Figure 3, we have an XSL transform that will take an XML version of our password database, and create a conventional /etc/passwd file. Along with simply reformatting the data into the desired format, it also does a selection to obtain all of our base entries (our normal users), along with the special entries we want on our AIX hosts. It does this by comparing the platform attribute for the desired targets. A trivial modification to this selection, and we can generate the passwd file for other platforms as needed.

```
<etc_passwd>
   <pw_entry platform="BASE">
      <username>finkej</username><unixuid>58220</unixuid><gid>4000</gid>
      <gecos>Jon Finke</gecos><homedir>/home/20/finkej</homedir>
      <shell>usr/bin/session</shell>
      <person_id>91325789</person_id><user_type>PRIMARY-EMP</user_type>
      <platform>BASE</platform>
   </pw_entry>
   <pw_entry platform="BASE">
      <username>hudsod</username><unixuid>58221</unixuid><gid>4000</gid>
      <gecos>Dave Hudson</gecos><homedir>/home/21/hudsod</homedir>
      <shell>usr/bin/session</shell>
      <person_id>91325792</person_id><user_type>PRIMARY-EMP</user_type>
      <platform>BASE</platform>
   </pw_entry>
   . . .
</etc_passwd>
```

Figure 2: XML version of /etc/passwd.

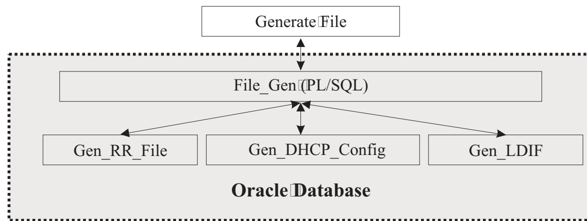

Figure 4: Original file model.

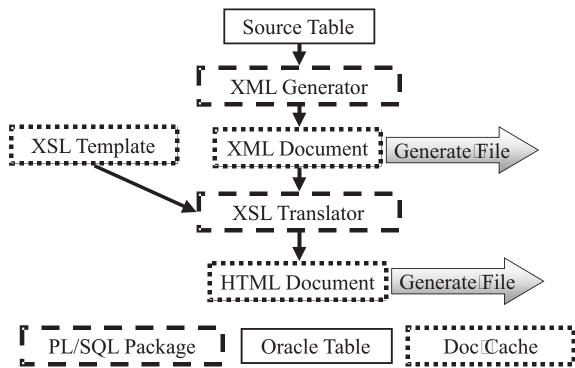

Figure 5: New file model.

Original Model

In the original model, seen in Figure 4, we would write a file specific PL/SQL package that "knew how" to generate the final file format. This package would be called by the File_Gen PL/SQL package that worked closely with the Generate_File program to extract the formatted information from the file specific package, and write it to the hosts file system (The Generate_File program can also read in files, read and write pipes, stdin, stdout and stderr). Together, these would handle most of the details of access control, error checking, version control and other details, allowing the file specific packages to concentrate on extracting the appropriate data and formatting it.

New File Model

As with the original model, we still write the appropriate PL/SQL code (the XML Generator in Figure 5) to generate a file from the desired data in the source table(s), only this time, instead of HTML or LaTeX, we generate it as an XML format document. We can, if desired, write this XML file out to an external file system, using the same Generate_File program as we used for the original model. All of the version control and selective generation features of the original approach are still available for use. The resulting XML files can then be used by the web developers to produce the desired outputs files, and made available for other services that need the data.

We don't need to stop with just the XML file. We can transform that XML document, inside of Oracle, with the appropriate XSL template into our desired HTML or LaTeX (or whatever) document and write out that file using the Generate_File program, just as we did before. This allows us to take full advantage of the version numbering of the original data, as well as the version number of the XSL template. One nice thing about this, is if the web developers provide a new style sheet via the XSL template, that forces the regeneration of all of the documents that used that style sheet. Note – this only requires that the original XML data documents be re-transformed, we do not need to repeat the original database queries used to generate them.

Version Control

In the original implementation, file version control was a one step process. We simply recalculated the version number based on the data in the database, and compared that to the version number in the file. If there was a difference, we regenerated the file with the new version number. With the addition of the XML support, we have at least two, and possibly three steps. As with the old model, we recalculate the version number based on the data, and then compare it with version number of the XML data stored in the document cache. At this point we now have an XML document, still stored in the database with a new version number. We can, if desired, write out this XML document to the external file system using the Generate_File program.

But although the XML file is nice, that is often not our desired objective. Instead of writing the XML document to a file, we can transform the XML document, via an XSL transform into a new document,

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text" indent="no" media-type="text/plain" standalone="yes" />
  <xsl:template match="/">
    <xsl:for-each select="/etc_passwd/pw_entry[platform='BASE' or platform='AIX']">
      <xsl:sort select="unixuid" order="ascending" case-order="lower-first"/>
      <xsl:value-of select="username"/>:*:<xsl:value-of select="unixuid"/>:
           <xsl:value-of select="gid"/>:<xsl:value-of select="gecos"/>:
           <xsl:value-of select="homedir"/>:
           <xsl:value-of select="shell"/><xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Figure 3: XSL transform for /etc/passwd.

possibly HTML or other format. This document is also still stored only in the database. It is however, a trivial matter to write this final form of the document to an external file. As an added twist, the version number of this transformed document is simply the greater of the version number of the base XML document and the version number of the XSL transform, which is yet another XML document stored in the database. In this way, when the XSL transform document is updated, all of the final documents that depend on that transform are now ''out of date'' and will be regenerated at the next available opportunity.

XML Document Cache

It quickly became obvious that we needed a way to store and reference XML documents easily within the database. Even if we were not concerned with the cost of regenerating the base XML documents every time we wanted to transform it, it is often desirable to be able to maintain a consistent data image. By generating and storing an XML document in the database, we are able to apply different transforms and know we are working on the same dataset.[2]

We had some specific applications in mind, and these helped drive our implementation. One of them was the our web directory of registered home pages. We actually have two different ones, the student home page directory and the Faculty/Staff home page directory. Aside from the title and the actual entries, these both use the same format. This was a case where we had the same schema but two different instances of that document. A second case was our departmental staff listing. For each department, we have a web page with all the people in that department. All use the same schema, but we have many different versions (one per department). In this case, we are looking at one instance (department staff list) of one schema (directory entry), but with many sub-documents, one for each department.

[2]Storing an XML document in Oracle is trivial. Oracle has a data type CLOB that can handle a document up to 2 GB in size (assuming you have the disk space available to handle it in the database). The XML stored packages provided by Oracle [10] allow manipulation of XML documents as CLOBs.

Another concern was maintaining access control on schemas, instances and documents. It is important to us to allow the web developers to update the templates used to generate the departmental directory web pages, yet not be able to change the templates used to generate the LaTeX used to produce the printed departmental directory. That access has to be granted to the text processing folks who manage the LaTeX world. Yet, both groups of people need to be able to read the same base documents. It is likely that our access control systems will become more fine grained over time.

Organizing Documents

The key element of any XML document is the schema. This defines what can be included in the document and how it is represented. Even if we don't actually validate the document, or even have the schema formally documented, it exists as a concept, and gives us a good starting point for organizing documents. In Figure 6 we have a schema, which would have basic information such as the name, a description, who created it and when, and so on. There still isn't any data at this level. In most cases, we will have a DTD document associated with this schema. This

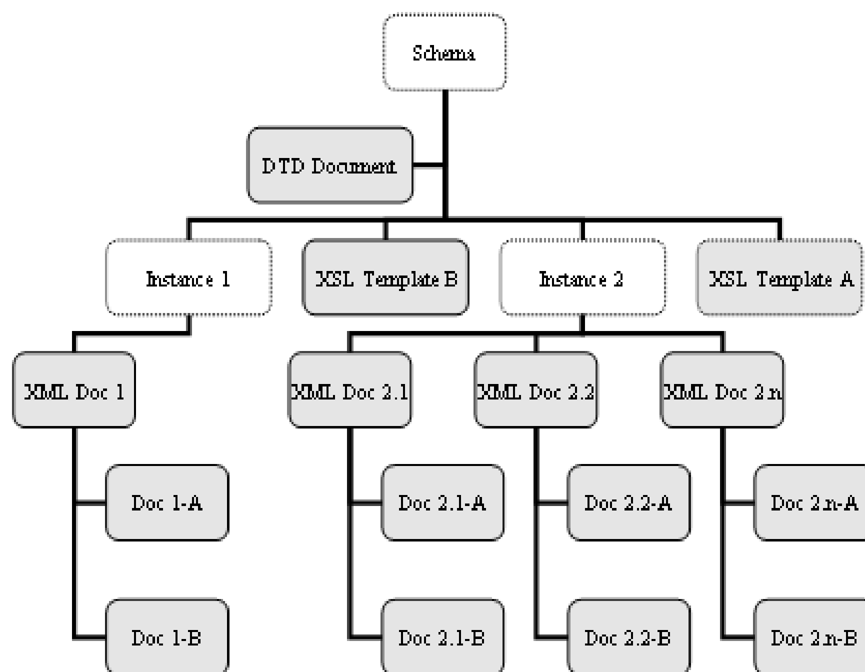

Figure 6: Schema model.

provides a guideline for both developers who want to generate XML documents as well as people who want to write XSL or other types of transforms.

Now that we have a schema, the next step is to define an instance of this schema. This will be an actual set of data, such as the password file or the departmental directory. This data will presumably need to be regenerated on a periodic basis. Since this is production data, we need to impose access controls on this instance of the schema; who can update it, and who can reference it. This instance may actually result in multiple sets of data – for instance, our employee directory is a set of documents, one per department. In this diagram, we have two instances, the first with just a single document and the second with multiple documents.

Since most people don't actually want to read XML files, the next step is to define an XSL template that will transform each of the XML data documents into the desired format, resulting in a second set of documents that can be written out as files or displayed via the web. We can provide additional translations to produce other formats. In the example of our staff directory, some departments want to have their own directory page on their web site. This allows them to use the official, most accurate, directory information and still maintain their desired look and feel. In Figure 6 we have two XSL templates (A and B), and we can apply those to each of the XML documents resulting in two output documents per input document.

These multiple transforms also allow us to include more data elements in the base XML documents that we might not want to include in the most broadly published documents. In the case of our employee directory, we include home address and telephone numbers, as well as cell phone and pager numbers. We do not want to put this into our regular online directory available to anyone via the Web, but we do want to include this in internal use department directories. By being able to control what elements are included as part of the XSL transform, all of our directory listings are able to have the most accurate and timely information.

Accessing Documents

When a document is stored in the document cache,[3] (see Table 1), we store a bunch of additional information about the document. This allows us to determine where a document originated, who put it there, and when. We also assign a unique document_id value that can be used to reference this document elsewhere in the system.

Extracting Documents

All documents stored in the document cache can be accessed via their Document_Id value. This allows us to have a couple of common routines used to extract

[3]My use of the word "cache" may be confusing to some folks. Caches are often short lived, whereas in this project, may last longer. An alternate description would be "Document Store."

Document_Id	Number	A unique identifier for this document
Schema_Id	Number	The identifier (primary key) of the schema that defines this document.
Instance_Id	Number	The identifier (primary key) of the schema instance that this document is part of. For a DTD document, this is the same as the Schema_Id.
Base_Doc_Id	Number	For a generated (XSL Transformed) document, this is the Document_Id of the base XML data.
Template_Id	Number	For a generated document, this is the document_Id of the XSL transform.
Doc_Code	varchar2(16)	For schema instances that have sub documents, this is the key that distinguishes the different documents from each other.
Doc_Type	varchar2(16)	This identifies the format of the document, such as "XML", "XSL", "DTD" and so on. This can be useful when displaying a document.
Doc_Title	varchar2(64)	This is a simple identifier for the document – useful for listing available documents without having to open each document up to determine what it is.
Description	varchar2(2000)	A place for a more in-depth description of a document than is allowed by the title field.
Entry_Method	varchar2(8)	A flag that indicates how this document was obtained, such as from an external URL, generated via direct query or other methods.
Ref_URL	varchar2(255)	The URL used to obtain this document if it was loaded via URL. Other entry_Methods may store source information such as the procedure or package name that generated the document.
Doc_Size	Number	The size of the document, in bytes.
When_Inserted	Number	The version number of this document, obtained for a system wide sequence number space.
Content	CLOB	The actual document, up to 2 GB in length.

Table 1: Document cache table.

a document, given the Document_Id. For both Generate_File and web display, we want to get the document one line at a time, until the end of the document. We did this with a PL/SQL function Get_Line as seen in Figure 7. This gets the document as a CLOB from the database, and breaks it out one line at a time.

```
Function Get_Line(doc_id in number)
return varchar2
is
    Next_Eol        Integer;
    Saved_Offset    Integer;
    Len             Integer;
begin
    if not Document_Open
    then
            Document_Open := True;
            Document := Get_Doc(Doc_Id);
            Document_Offset := 1;
    end if;
    next_eol := Dbms_Lob.Instr(Document,
              chr(10), Document_Offset);
    if next_eol = 0
    then
        Document_Open := False;
        return null;
    end if;
    Len := Next_Eol - Document_Offset ;
    Saved_Offset := Document_Offset;
    Document_Offset := Next_Eol + 1;
    Return Dbms_Lob.Substr(Document,
                    Len, Saved_Offset);
end Get_Line;
```

Figure 7: Get_Line function.

Generating XML from a Simple SQL Query

One of the handy PL/SQL packages that comes with Oracle 8i, allows you to define a SQL query, and pass it to a routine, and it will execute the query and return an XML document. Now it is possible that the existing Oracle table layout does not directly provide you with the exact thing needed for your XML, so one approach is to create an view.[4] To create the XML version of our password file, we created the view shown in Figure 8. This view extracts all of our normal userids (based on the SOURCE attribute) and then combines that list with the "SPECIAL" userids. For normal users, we derive the home directory from the UID and username, and add a default shell and a default group id if none is supplied. For the special ids, we generate those same fields if none are provided in the Special_Ids table.

The next step is to turn that into XML. To do this, we packaged up the Oracle routines, with some of our own to give us the code seen in Figure 9. We simply set up the query of the view, and then pass it to the XML_Cache_Maint package which will execute the query, turn it into XML, store it and return the Document_Id to us. This gives us the XML seen in Figure 2. The first bunch of parameters are used to tie into the document cache, defining the schema name and instance, adding a title and a description. Since we are providing this as a single document (no sub documents), there is no Doc_Code value. The next set of parameters are passed to Oracle and tell it how to label the XML. The gen_dtd tells it to include the DTD at the start of the document, and the last parameter tells the document cache to create a new schema if one does not already exist. This last argument is to help bootstrap the system into place and should not be needed for production file generations.

Generating XML from a Not So Simple SQL Query

The method of generating an XML document from a SQL query can handle a lot of basic XML generation needs, but at times, we need more complex documents. Consider the XML required to generate the /etc/group file (see Figure 11). The DTD for this (see Figure 10) is similar to that for /etc/passwd, but with the change that one of the elements (userlist) is in fact a subtype with multiple elements allowed.

As with the /etc/passwd example, the entire document is wrapped in a <etc_group> tag, and has a number of <group_entry> entries. The expected group fields (group name, group password, group id) as well as a <platform> attribute that we use to define the type of system for which this entry is intended. A group can have zero, one or more than one members in it. To handle these, we have the field <userlist> which will hold as many usernames (all tagged with <userlist_item> tags). You will note in the second entry for the group user, the user list entry is written as <userlist/>. This indicates an empty list.

[4] A view is a predefined query that can operate on one or more base table and be referenced much like a table.

```
Select Username, unixuid "UNIXUID", nvl(unixgid,4000) GID,
       Public_Personal_Info GECOS, 'usr/bin/session' SHELL,
       '/home/' || ltrim(to_char(mod(unixuid,100),'00')) || '/' ||
       username HOMEDIR, owner PERSON_ID, source USER_TYPE, 'BASE' PLATFORM
  from logins
 where (    substr(source,1,7) = 'PRIMARY' or source = 'SECONDARY')
Union
Select L.username, l.unixuid, nvl(l.unixgid,4000),
       L.public_personal_info,nvl(SI.Shell,'usr/bin/session'),
       nvl(SI.Home_Dir,'/home/' || ltrim(to_char(mod(L.unixuid,100),'00')) ||
                 '/' || l.username), owner, source, nvl(SI.Platform,'BASE')
  from Logins L, special_Ids si
 where l.source = 'SPECIAL'
   and l.username = si.username
```

Figure 8: Oracle view: etc_passwd.

```
<!DOCTYPE etc_group [
<!ELEMENT etc_group (group_entry)*>
<!ELEMENT group_entry (gname, gpasswd?,
                gid, platform?, userlist?)>
<!ATTLIST group_entry platform CDATA
                            #REQUIRED>
<!ELEMENT gname (#PCDATA)>
<!ELEMENT gpasswd (#PCDATA)>
<!ELEMENT gid (#PCDATA)>
<!ELEMENT platform (#PCDATA)>
<!ELEMENT userlist (userlist_item)*>
<!ELEMENT userlist_item (#PCDATA)>
]>
```

Figure 10: DTD for /etc/group .

To generate this data, we need to access data from three tables. The primary table is the groups table that holds the basic information about the group (aside from the membership). We then need to reference the group_members table to get the list of members for each group and finally the logins table to get the actual usernames. Fortunately, Oracle provides us with a way of doing that.

To do this, we first define a type XML_Etc_Group_Userlist as a TABLE OF VARCHAR2(8). We then create an object view of the groups as shown in Figure 12. This creates a view of a complex object that includes a sub-query for each row. When the view is referenced, for each row (group), it obtains all of the members of that list, and converts those usernames. That entire list is returned as the single element userlist. When we pass a "SELECT * FROM XML_ETC_GROUP" query to Cache_Query routine, the XML in Figure 11 is generated.

Complex Generation

When the generation of the XML becomes too complex for the query method, even with the complex

```
Query := 'Select username, unixuid, gid, gecos, homedir,'
      || '       shell, person_id, user_type, platform'
      || '  from XML_Etc_Passwd order by uid';

did := XML_Cache_Maint.Cache_Query(
        query => Query,
        Schema_name => 'etc_passwd',
        Schema_instance => 'General RCS',
        d_title => 'ETC Passwd',
        d_code => Null,
        d_desc => 'An XML version of the user base',
        row_tag => 'PW_Entry',
        row_set_tag => 'etc_passwd',
        row_id_attr_name => 'Platform',
        row_id_attr_value => 'PLATFORM',
        gen_dtd => True,
        create_schema => True);
```

Figure 9: Create and cache XML from a query.

```
<etc_group>
   <group_entry platform="rs_aix">
      <gname>adm</gname><gid>4</gid><platform>rs_aix</platform>
      <userlist>
         <userlist_item>bin</userlist_item>
         <userlist_item>adm</userlist_item>
      </userlist>
   </group_entry>
   <group_entry platform="BASE">
      <gname>user</gname><gpasswd>*</gpasswd><gid>4000</gid><platform>BASE</platform>
      <userlist/>
   </group_entry>
  ...
</etc_group>
```

Figure 11: XML Data for /etc/group .

```
Select Group_Name, Group_Passwd, Group_Id, group_index, nvl(platform,'BASE'),
       cast ( multiset ( select username
                           from group_members gm, logins l
                          where gm.group_index = g.group_index
                            and gm.login_id = l.login_id
                  ) as XML_Etc_Group_Userlist
         ) as userlist
  from groups g
```

Figure 12: XML_Etc_Group view definition.

views, we can fall back to building the XML element by element. Although we could simply build it up as a string using character manipulation, Oracle provides some additional packages that can assist.

There are several approaches for manipulating XML documents, and one of those is the Document Object Model (DOM). With the DOM model, the XML document is represented by a tree structure built in memory. This can be manipulated in a number of ways, and then eventually exported as an XML document. Oracle has a Java implementation of the DOM routines, and they have provided a PL/SQL interface to them as well. In Figure 13, we have the main procedure that we use to generate our departmental directory.

This creates a new document using the DOM routines, assigns a root node to it, and then calling some existing directory code (Get_Univ_Web_List) we get a list of all departments in the directory. We pass each entry to a formatting routine (Gen_Entry – Figure 14) which formats each entry into a node on the XML tree. Since a given entry may have sub entries, we check each node for children, and append appropriate directory entries where needed. When we reach the end of the list of departments, we take the now fully populated XML tree and save it in the document cache. This XML document is now available for transformation into other formats or to be written as XML for external processes to manipulate.

Although this approach of building up an XML document one element at a time may appear to be tedious, it is actually less tedious than the previous approach of generating HTML directly. In addition, by moving the details of the presentation into the XSL transformation, the resulting XML code is simpler than the HTML it replaces. Another nice win with the generation of XML is that since the transformation can ignore extra data elements, it makes it much easier to write generic XML generation routines that include everything and let the XSL transforms sort it out. This eliminates a lot of duplicate or very similar code and ensures that logic changes apply everywhere.

```
doc := XMLDOM.newDOMDocument;   -- Get a fresh, empty document
root := XMLDOM.createElement(doc,'Institute_Directory'); -- Create a root node
dmy := XMLDOM.appendChild(XMLDOM.makeNode(doc),root);   -- And link it in
loop
        R := Generate_FacStaff_Dir.Get_Univ_Web_List;
        exit when r.name is null;
        Gen_Entry(root,R);
end loop;
did := XML_Cache_Maint.Cache_Dom_Doc(
        Document => doc,
        Schema_name => 'Department_Phone_List',
        Schema_instance => 'Institute Telephones',
        d_title => 'Institute Office Directory',
        d_code => Null,
        d_desc => 'Directory of phone nums/web pages for Inst. Offices.',
        d_type => 'XML');
Commit;
XMLDOM.freeDocument(doc);
```

Figure 13: Directory generation main procedure.

```
procedure Gen_Entry(node in XMLDOM.DOMNode, R in Generate_FacStaff_Dir.Univ_Tel_Rec) is
        This_Node       XMLDOM.DOMNode;
        dmy     XMLDOM.DOMNode;
        Child   Generate_FacStaff_Dir.Univ_Tel_Rec;
begin
       -- Create the entry node and add it to the list
        This_Node := XMLDOM.makeNode(XMLDOM.CreateElement(Doc,'Dir_Entry'));
        dmy := XMLDOM.appendChild(node,This_Node);
        -- Populate the top level records in the node
        XML_DOM_Utils.Make_And_Append(This_Node,'Name',R.Name);
        XML_DOM_Utils.Make_And_Append(This_Node,'TELEPHONE',R.Phone);
        XML_DOM_Utils.Make_And_Append(This_Node,'URL',R.Url);
        XML_DOM_Utils.Make_And_Append(This_Node,'EMAIL',R.Email);
        XML_DOM_Utils.Make_And_Append(This_Node,'FAX',R.Fax);
        -- add in children ...
        Loop
                Child := Generate_FacStaff_Dir.Get_Univ_Tel_Children(R.Orgn);
                exit when Child.Name is null;
                Gen_Entry(This_Node, Child);
        end Loop;
```

Figure 14: Directory generation Gen_Entry procedure.

Storing from a URL

While we are often able to generate our XML data files from data stored in the database, we sometimes need to load in other files (such as XSL template files) from other places. Another interface provided by Oracle, is the ability to take a URL, query the web, parse the XML document and return DOM format XML document. We wrapped that in our own routine (Figure 15) that then stores the resulting document in the document cache. Since we save the URL used in the document cache, we also have a Refresh_Url routine that will look up the URL from the document cache, repeat the query, and if the resulting document is valid, save the updated version in the cache.

```
function Cache_Url(
      url in varchar2,
      Schema_name in varchar2,
      Schema_instance in varchar2,
      d_title in varchar2,
      d_code in varchar2,
      d_desc in varchar2,
      d_type in varchar2)
return number;  -- Document_Id
```

Figure 15: Read and cache XML from a URL.

```
function Transform_And_Cache(
      Schema_name in varchar2,
      Document_Instance in varchar2,
      Transform_Document in varchar2,
      d_title in varchar2,
      d_code in varchar2,
      d_desc in varchar2,
      d_type in varchar2)
return number;  -- Document_Id
```

Figure 16: Transform and cache XML.

Unlike some of the other XML_Cache_Maint routines, this particular routine is generally only called from our web interface to the document cache. Since the URL query might fail, either due to connectivity problems[5] or the file obtained via the URL might no longer be a valid XML document. Longer term, this facility has much potential for transferring data into the database from foreign systems with the proper detection and reporting of failures.

Translation and Storing

In order to take advantage of the use of XML to separate data extraction from the presentation, and still take advantage of the version control support provided by the original Generate_File program, we need to be able to apply an XSL transform to an XML document within the database, and store the resulting document in the document cache. To this end, we have again wrapped the Oracle routines, with a function of our own (see Figure 16) to handle the translation and store the results back in the document cache.

[5]When Oracle processes this request, the database server attempts to make an HTTP connection to the specified web server.

With this version of the function, we supply the schema name, which is shared by both the Document Instance (the base XML document) and the Transform Document(the XSL transform document, which is just another XML document.) We also provide the document code for those instances with multiple documents. There is a version planned that will iterate over all of the sub documents in an instance, and regenerate all that are out of date (version control) automatically.

Conclusions and Futures

The ability to separate data extraction (like for the online directory) from the presentation has really helped in maintaining a consistent look and feel for the University web pages; prior to this work, the online directory pages lagged three to six months behind the rest of the web sites in their appearance. We still have a number of places where we generate HTML for both static pages, and for dynamic applications. Many of these may get rewritten to generate an intermediate XML format before the final HTML presentation.

Another area of exploration will be with DocBook [12], which is a markup standard markup for computer documentation and technical books developed by the joint efforts of Hal Computer Systems International, Ltd., and O'Reilly and Associates, Inc. This would be a good addition to the *Service Trak* [5] project to allow us to integrate service documentation with the current state of services at our site. XML will also dovetail nicely into Service Trak for the generation of configuration files for Big Brother and other service monitoring packages.

As use of the XML file generation grows, I expect we will be tuning and expanding the access control options for every stage of the document generation process. I also expect to make some refinements in the way we set up new targets for the Generate_File program. At this point, each new target requires a new PL/SQL package. With the Cache_Query support, a number of file generation targets can be reduced to a simple query and a few parameters. This will make it much easier to extract data from the database for external applications.

References and Availability

This project is part of (but not dependent on) the Simon system, an Oracle based system used to assist in the management of our computer accounts [7], enterprise white pages [4], printing configuration [3], All source code for the Simon system, including Generate File, is available on the web. See http://www.rpi.edu/campus/rpi/simon/README.simon for details. At present we have both AIX and Solaris versions of the Generate_File program in production and efforts are underway to finish a Java version. In addition, all of the Oracle table definitions as well as PL/SQL package source are available at http://www.rpi.edu/campus/rpi/simon/misc/Tables/simon.Index.html .

Some of the source code examples in this paper have been modified from the actual production code. I would suggest that if you are interested in this work, that you review the current source code available from the above URL.

Other Environments

XML has sparked a lot of interest from many people and as a result, there are a lot of XML and related resources available. I have found the SGML module for Emacs handy and there are many other editing tools that understand XML and XSL file formats. Along with editing environments, there are a number of programming interfaces and packages available, many of them for free. Aside from the PL/SQL packages provided by Oracle, there are Java, C++, perl and others.

Acknowledgements

I would like to thank Mario Obejas for his shepherding of this paper, as well as Jeff R. Allen, the LISA copy editor. I also want to thank Rob Kolstad for his excellent (as usual) job of typesetting this paper. Thanks also to Arlen Johnson and Kevin Bishop of Communication and Collaboration Technologies at Rensselaer (these are those wacky web guys who actually make good looking web pages) for their help with the XSL and XML design, and Alan, Andy, Bick and Kelly who helped review this paper.

Author Biography

Jon Finke graduated from Rensselaer in 1983 with a BS-ECSE. After stints doing communications programming for PCs and later general networking development on the mainframe, he then inherited the Simon project, which has been his primary focus for the past 12 years. He is currently a Senior Systems Programmer in the Networking and Telecommunications department at Rensselaer, where he continues integrating Simon with the rest of the Institute information systems. When not playing with computers, you can often find him building or renovating houses for Habitat for Humanity, as well as merging a pair of adjacent row houses into one. Reach him via USMail at RPI; VCC 319; 110 8th St; Troy, NY 12180-3590. Reach him electronically at finkej@rpi.edu. Find out more via http://www.rpi.edu/˜finkej.

References

[1] Adler, Sharon, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Tony Graham, Eduardo Gutentag, Eduardo Gutentag, Scott Parnell, Scott Parnell, and Steve Zilles, *Extensible Stylesheet Language (xsl) Version 1.0*, http://www.w3.org/TR/xsl, 2001.

[2] Bosak, Jon, *Extensible Markup Language (xml) version 1*, http://www.w3.org/TR/REC-xml, 1997.

[3] Finke, Jon, "Automating Printing Configuration," *USENIX Systems Administration (LISA VIII) Conference Proceedings*, pp. 175-184, USENIX, San Diego, CA, September, 1994.

[4] Finke, Jon, "Institute White Pages as a System Administration Problem," *The Tenth Systems Administration Conference (LISA 96) Proceedings*, pp. 233-240, USENIX, Chicago, IL, October 1996.

[5] Finke, Jon, "Automation of Site Configuration Management," *The Eleventh Systems Administration Conference (LISA 97) Proceedings*, pp. 155-168, USENIX, San Diego, CA, October, 1997.

[6] Finke, Jon, "An Improved Approach to Generating Configuration Files from a Database," *The Fourteenth Systems Administration Conference (LISA 2000)*, pp. 29-38, USENIX, New Orleans, LA, December, 2000.

[7] Finke, Jon, "Embracing and Extending Windows 2000," *The Sixteenth Systems Administration Conference (LISA 2002)*, USENIX, November, 2002.

[8] Finke, Jon, "Process Monitor: Detecting Events That Didn't Happen," *The Sixteenth Systems Administration Conference (LISA 2002)*, pp. 145-153, USENIX, November, 2002.

[9] Goossens, Michal and Sebastian Rahtz, *The LaTeX Web Companion*, Chapter 6, "Tools and Techniques for Computer Typsetting," Addison-Wesley, May 1999.

[10] Higgins, Shelly, *Oracle8i Application Developer's Guide – XML*, September, 2000.

[11] Portfolio, Tom, *PL/SQL Release 8 User's Guide and Reference*. Oracle Corporation, Part Num. A58236-01, December, 1997.

[12] Walsh, Norman and Leonard Muellner, *DocBook: The Definitive Guide*, O'Reilly and Associates, ISBN 1-56592-580-7, Sebastopol, CA, October, 1999.