XML AND ORACLE: A HOW-TO GUIDE FOR PL/SQL USERS

Eashwar Iyer, Quovera

What's the best way of exchanging data between different sources without worrying about how the receiver will use it? What's the best way of creating documents with the right content without worrying how it should be displayed on the web and then able to display them with all the flexibility one could get? Welcome to the world of XML and its family of technologies.

This whitepaper is aimed at understanding XML and related topics viz. XSL, DTD, DOM, SAX and Schemas. It also looks at some of the products and tools from Oracle that supports XML through PL/SQL. Please note that a downloadable version of this paper and the associated presentation are available at www.quovera.com/forum/index.html

XML AND THE RELATED TOPICS

XML

XML stands for eXtensible Markup Language. In contrast to HTML that describes visual presentation, XML describes data in an easily readable format but without any indication of how the data is to be displayed. It is a database-neutral and device-neutral format. Since XML is truly extensible, rather than a fixed set of elements like HTML, use of XML will eventually eliminate the need for browser developers and middle-ware tools to add special HTML tags (extensions). Listing 1 is an example of a simple XML document.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="Employee.XSL"?>
<Employees>
 <Empl id="1">
      <FirstName>
        Chuck
      </FirstName>
      <LastName>
        White
      </LastName>
      <Dept>
        Finance
      </Dept>
 </Empl>
</Employees>
```

Listing 1: Employee.xml

XSL

As explained earlier, XML is more focussed on defining the data, therefore, we need a mechanism to define how this data should be displayed in browsers, cell phones or any other such devices. This is exactly what *XSL (eXtensible Style Language)* does. It defines the rules to interpret the elements of the XML document.

XSL at its most basic provides a capability similar to a "mail merge." The style sheet contains a template of the desired result structure, and identifies data in the source document to insert into this template. This model for merging data and templates is referred to as the *template-driven model* and works well on regular and repetitive data. Listing 2 is an example of an XSL document for the XML document shown in Listing 1.

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
```

```
  <xsl:template match="/">
    <HTML>
      <BODY>
      <h1>Employee Details</h1>
        <xsl:for-each select="Employees/Empl">
            <b>Empl #
               <xsl:value-of select="@id" /> </b>
            <i>First Name :
               <xsl:value-of select="FirstName" /> </i>
            <i>Last Name  :
               <xsl:value-of select="LastName" /> </i>
            <i>Dept       :
               <xsl:value-of select="Dept" /> </i>
         </xsl:for-each>
      </BODY>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```

Listing 2: Employee.xsl

Figure 1 shows the way the browser interprets the Employee.xml document when combined with the Employee.xsl document.

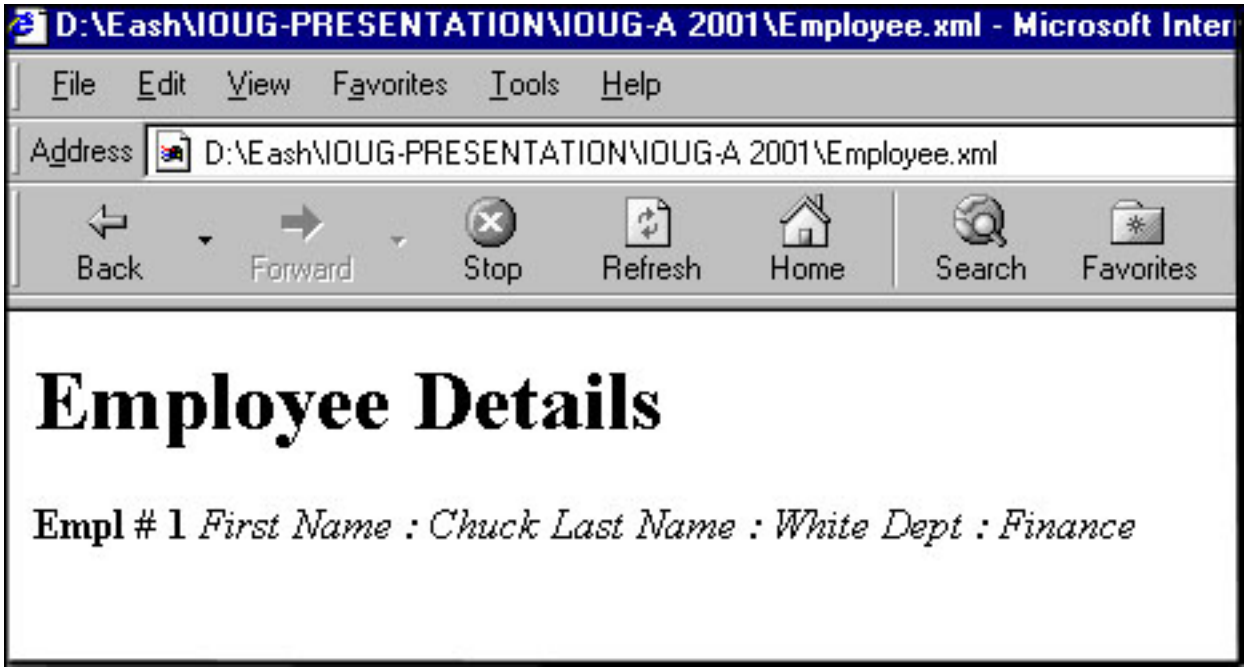

Figure 1: Output in browser when Employees.xml is called.

DTD

DTD (Document Type Definition) is a set of rules or grammar that we define to construct our own XML rules (also called a "vocabulary"). In other words, a DTD provides the rules that define the elements and structure of our new language.

This is comparable to defining table structures in Oracle for a new system. As we define the columns of a table, determine the datatypes of the columns, determine if the column is 'Null' allowed or not, the DTD defines the structure for the XML document. Listing 3 is an example of a basic DTD. The detailed syntax of DTD is covered later in the paper.

```
<Employees>
 <Empl>
      <FirstName>
      </FirstName>
      <LastName>
      </LastName>
      <Dept>
      </Dept>
 </Empl>
</Employees>
```

Listing 3: Employee DTD

DOM

The Document Object Model (DOM) is a simple, hierarchical naming system that makes all of the objects in the page, such as text, images, forms etc accessible to us. It is merely a set of plans that allow us to reconstruct the document to a greater or lesser extent.

By definition, a complete model is one that allows us to reconstruct the whole document down to the smallest detail. An incomplete DOM is anything less than that.

For the reader's information, the W3 DOM recognizes seventeen types of node objects for XML: Attribute, CDATASection, Comment, DOMImplementation, Data, Document, DocumentType, DocumentFragment, Element, Entity, EntityReference, NamedNodeMap, Node, NodeList, Notation, ProcessingInstruction, Text

For a detailed description of other node types, the reader is encouraged to visit the W3 web site at http://www.w3.org/TR/WD-DOM/object-index.html.

SAX

Simple API for XML (SAX) is one of the two basic APIs for manipulating XML. It is used primarily on the server side because of its characteristics of not storing the entire document in memory and processing it very fast. However, SAX should be used mainly for reading XML documents or changing simple contents. Using it to do large-scale manipulations like re-ordering chapters in a book or any such activities will make it extremely complicated, not that it cannot be done.

SCHEMA

It's a mechanism by which rules can be defined to govern the structure and content relationship within a document.

XML Schema Structures specifies the XML Schema definition language, which offers facilities for describing the structure and constraining the contents of XML 1.0 documents. The schema language, which is itself represented in XML 1.0 and uses namespaces, substantially reconstructs and considerably extends the capabilities, found in XML 1.0 document type definitions (DTDs). This specification depends on XML Schema Part 2: Datatypes.

XML Schema Datatypes is part 2 of the specification of the XML Schema language. It defines facilities for defining datatypes. The datatype language, which is itself represented in XML 1.0, provides a superset of the capabilities found in XML 1.0 document type definitions (DTDs) for specifying datatypes on elements and attributes.

NAMESPACES

With XML namespaces developers can qualify element names uniquely on the Web and thus avoid conflicts between elements with the same name. The association of a Universal Resource Identifier (URI) with a namespace is purely to ensure that two elements with the same name can remain unambiguous; no matter what the URI points to.

WELL-FORMED VS. VALID XML DOCUMENTS:

Well-formed documents are those that conform to basic rules of XML such as a) the document must have only one root element, b) it must have start and end tags for every element etc.

Valid documents are not only well-formed but also have been validated against a DTD (or a schema). A parser usually does the validations.

XML IN ORACLE

In order to see and appreciate the implementation of XML in Oracle, we need to have the necessary products and components installed. The next section briefly looks at the products that are required.

A ROUND TRIP EXAMPLE

WHAT DO WE WANT TO ACHIEVE?

To enjoy all the benefits provided by XML (and XSL, DTD etc.), the least we should be able to do are:

- Read data from the database and convert them into an XML document.
- Output the XML documents in the appropriate device (we will restrict ourselves to displaying the output in a browser).
- Read XML document and insert the data contained in it into the table in the database.

For any real life application, the first step would be to design the database table/s and the corresponding DTD. Thereafter, an XSL document will be required for displaying the resultant XML document meaningfully. The application code to do all these manipulations will then follow.

To start with, lets consider a "Zipcodes" database table with the structure as shown in Table 1:

Column_Name	Data_Type	Width
State_Abbreviation	Character	2
ZipCode	Numeric	5
Zip_Code_Extn	Numeric	4
City	Character	50

Table 1: Zipcodes table structure

MODELING THE TABLE STRUCTURE

Assuming that the table design is fine, our first step will be to create a simple DTD that mirrors its structure. We've already listed the fields that make up each record in the "Zipcodes" table. There are some other rules we can state about the table:

- The table name is "Zipcodes".

- Each record in the "Zipcodes" table represents a complete mapping of zip code, the extra four digits of zip code, the city name and the state abbreviation.

As a first pass at a DTD, we'll create the tags <Zipcodes>, <mappings>, etc., specifying the relationships among items we just outlined. Before we do that, we'll discuss the basics of DTD syntax.

DTD Basics

Each statement in a DTD uses the <!XML DTD> syntax. This syntax begins each instruction with a left angle bracket and an exclamation mark, and ends it with a right angle bracket.

Defining a Document Element

The document element, the outermost tag, will be the <Zipcodes> tag:

```
<!ELEMENT Zipcodes (mappings)+>
```

The element declaration defines the name of the tag ("Zipcodes"), and the content model for the tag ("mappings").

The + notation after the content model above means the <Zipcodes> tag must contain one or more <mappings> tags. There are other such notations in XML as follows:

* - The content can appear 0 or any number of times.

? - The content can appear 0 or once.

[none] - The content must appear once as shown.

First Cut DTD

Listing 4 shows our first-cut DTD. The first line shows the name of the table and shows content model as described in the previous paragraph. The second line shows all the column names in a row in the table. It does not show any occurrence indicator. It means the elements must occur in the same sequence and only once. The next four lines show the columns of our table within tags. The #pcdata keyword means the tags contain 'parsed character data'.

```
<!ELEMENT Zipcodes (mappings)+>
<!ELEMENT mappings (state_abbreviation, zipcode, zip_code_extn, city)>
<!ELEMENT state_abbreviation (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>
<!ELEMENT zip_code_extn (#PCDATA)>
<!ELEMENT city (#PCDATA)>
```

Listing 4: First-cut DTD

Extended DTD

There are many other keywords for defining DTDs. Let us look at a few of them here.

The "Attribute" keyword allows us to define the attributes that can appear inside a tag as well as the kinds of data the attributes can contain. The following extension to our earlier DTD, Listing 3, shows this.

```
<!ELEMENT Zipcodes (mappings)+>
<!ELEMENT mappings (state_abbreviation, zipcode, zip_code_extn, city)>
<!ELEMENT state_abbreviation (#PCDATA)>
<!ATTLIST state_abbreviation
      state (AL | AK | AZ | AR | AS | CA | CO | CT) #REQUIRED
>
<!ELEMENT zipcode (#PCDATA)>
<!ATTLIST zipcode
      zipcode CDATA #REQUIRED
```

```
>
<!ELEMENT zip_code_extn (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ATTLIST city
      city CDATA #REQUIRED
>
```

Listing 5: Extended DTD

The "#REQUIRED" keyword in the attribute definition means that this attribute must be coded for each and every <state-abbreviation>, <zipcode> and <city> tag in our document. For attributes that are not required, we can use the #IMPLIED keyword.

DEFINING THE XSL DOCUMENT FOR THIS

The XSL document in Listing 6 defines how the browser will handle our XML document (based on the above DTD). In this example we will refer to the standards defined for Internet Explorer 5.x. This may not work well for Netscape.

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <HTML>
      <BODY>
        <TABLE BORDER="2">
          <TR>
            <TD>Zipcode</TD>
            <TD>Zip Code Extn</TD>
            <TD>City</TD>
            <TD>State Abbreviation</TD>
          </TR>
          <xsl:for-each select="Zipcodes/mappings">
            <TR>
              <TD><xsl:value-of select="ZIPCODE"/></TD>
              <TD><xsl:value-of select="ZIP_CODE_EXTN"/></TD>
              <TD><xsl:value-of select="CITY"/></TD>
              <TD><xsl:value-of select="STATE_ABBREVIATION"/></TD>
            </TR>
          </xsl:for-each>
        </TABLE>
      </BODY>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```

Listing 6: XSL Document for the zipcodes XML

This XSL document defines a table with double-lined borders for displaying the XML "zipcodes" document. Note the headers for the columns and the columns themselves are defined in a different sequence than the one in the XML document and the corresponding table. This XSL document is generic and will accommodate any number of rows ("mappings") in the XML document. We will see this XSL document in action very soon.

IMPLEMENTING THE EXAMPLE IN ORACLE

I have created the "zipcodes" table in the database and added three rows in it as shown in Listing 7.

```
SQL> select * from zipcodes;

ST ZIPCO ZIP_ CITY
-- ----- ---- ----------------------------
CA 95123 6111 San Jose
CA 95234      Sunnyvale
AK 72701      Fayetteville
```

Listing 7: Output of select statement on the zipcodes table.

CREATING AN XML DOCUMENT FROM THE TABLE

The rows in the table can be converted in the form of an XML document using the Oracle XSU and Oracle XML Parser utilities (refer earlier sections to get download and installation information).

The PL/SQL code in Listing 8 is for creating an XML document from our "zipcodes" table. This code uses the UTL_FILE package to write the XML document in the file system.

```
Rem A similar code is available on technet.oracle.com
Rem I have revised and used it here for demonstrating the creation of the XML
Rem document. Remember to give access to the necessary directory (D:\test in this
Rem example) in the file system in init.ora


declare
 xmlString CLOB := null;
 amount integer:= 1000;
 position integer := 1;
 charString varchar2(1000);
 fileHandle UTL_FILE.FILE_TYPE;

begin

--we want the result document root to be "Zipcodes"
--to follow our DTD structure
  xmlgen.setRowsetTag('Zipcodes');

--we want the row element to be named "mappings" to follow our DTD structure
  xmlgen.setRowTag('mappings');

--open the file in "write" mode
  fileHandle := utl_file.fopen('d:\test','XML_FOR_ZIPCODES.XML', 'w');

--set the ERROR tag to be ERROR_RESULTS
  xmlgen.setErrorTag('ERROR_RESULT');

--set the id attribute in the ROW element to be Record - so that it shows the number
--of records fetched
  xmlgen.setRowIdAttrName('Record');

--do not use the null indicator to indicate nullness
  xmlgen.useNullAttributeIndicator(false);

--attach the stylesheet to the result document
  xmlgen.setStyleSheet('XSL_FOR_ZIPCODES.XSL');

--This gets the XML out - the 0 indicates no DTD in the generated XML document
--a value of 1 will provide a DTD description in the XML document
  xmlString := xmlgen.getXML('select * from scott.zipcodes',0);
```

```
--Now open the lob data..
  dbms_lob.open(xmlString,DBMS_LOB.LOB_READONLY);
  loop
    -- read the lob data
    dbms_lob.read(xmlString,amount,position,charString);
    utl_file.put_line(fileHandle, charString);
    position := position + amount;
  end loop;

  exception
  when no_data_found then
    -- end of fetch, free the lob
    dbms_lob.close(xmlString);
    dbms_lob.freetemporary(xmlString);
    xmlgen.resetOptions;
    utl_file.fclose(fileHandle);
  when others then
    xmlgen.resetOptions;
end;
/
```

Listing 8: Code for creating the XML document from rows in the zipcodes table

Important Note: The above code will create an XML document in the file system. In order to make it display the output in a browser, this code can be modified to use the 'htp' functions and the call can be made from a Web Server.

DISPLAYING THE XML DOCUMENT IN A BROWSER

For understanding how the XML and XSL work together, we can open the XML document from the browser (Internet Explorer, for this example). Ensure that the XSL document is in the same directory as the XML document. Figure 1 shows the output in IE.

Note: Alternatively, we can take the XML document and parse it into a DOM (Document Object Model) tree. Building a DOM tree is typically the first step in processing an XML document. Once the DOM tree is built, we can look at the information in it and convert it into HTML. The DOM tree is a very useful data structure that allows us to manipulate the contents of the XML document. For readers who are interested in trying this out, XMLDOM and related packages of Oracle XML Parser for PL/SQL can be used.

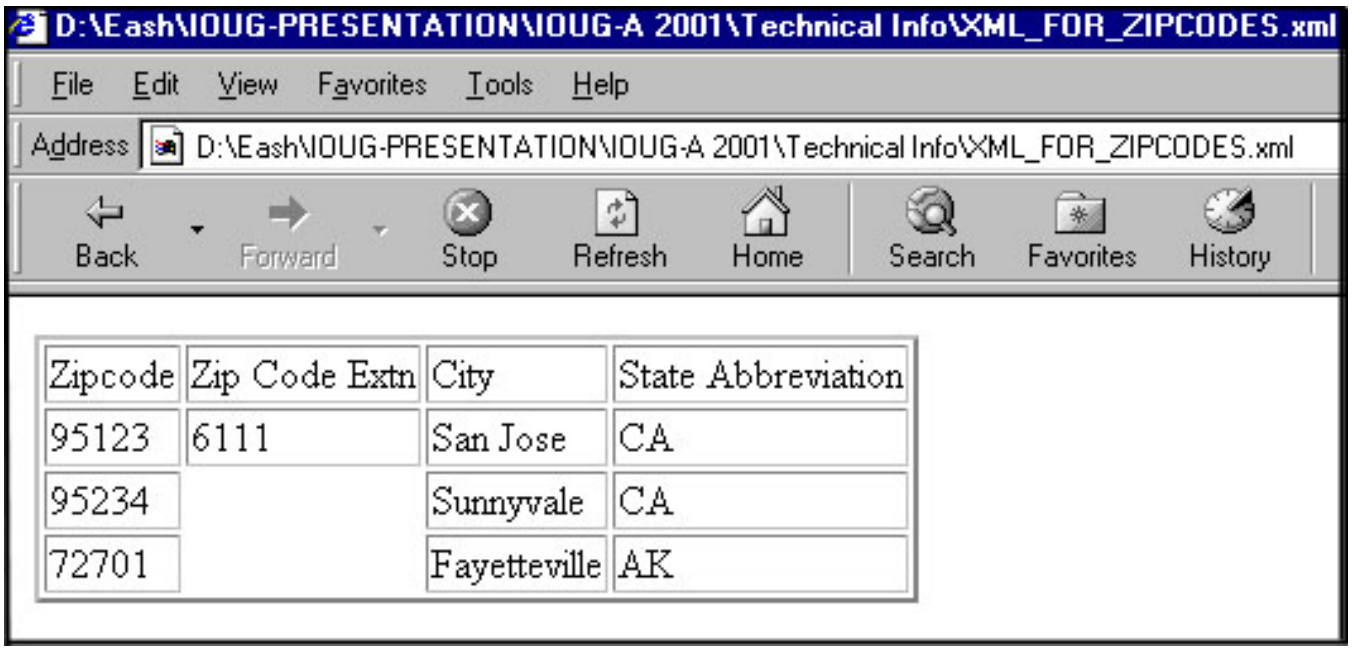

Figure 2: Output in browser when XML_FOR_ZIPCODES.XML is called.

READING AN XML DOCUMENT INTO THE DATABASE

Again, the Oracle XSU and Oracle XML Parser utilities come handy in doing this. Listing 9 shows the code used for reading an XML document and inserting the data into our "zipcodes" table. This code uses the UTL_FILE package to open the XML document from the file system. The XML document that is read is shown in Listing 10. The records in the "zipcodes" table, after executing this code, are shown in Listing 11. Note that two records have been added to the table from the XML document.

```
Rem PL/SQL code to read an XML file and write the data to the database table
Rem "zipcodes"
Rem It uses UTL_FILE package to access the XML file from the file system
Rem Remember to give access to the directory (D:\test in this example) in the file
Rem system in init.ora
Rem Remember to set the serveroutput to on

declare
 charString varchar2(80);
 finalStr varchar2(4000) := null;
 rowsp integer;
 v_FileHandle UTL_FILE.FILE_TYPE;

 begin

-- the name of the table as specified in our DTD
  xmlgen.setRowsetTag('Zipcodes');
-- the name of the data set as specified in our DTD
  xmlgen.setRowTag('mappings');

-- for getting the output on the screen
     dbms_output.enable(1000000);

-- open the XML document in read only mode
     v_FileHandle := utl_file.fopen('d:\test','XML_NEW_CITIES.XML', 'r');
```

```
     loop
      BEGIN
        utl_file.get_line(v_FileHandle, charString);
        exception
           when no_data_found then
           utl_file.fclose(v_FileHandle);
           exit;
       END;
         dbms_output.put_line(charString);
         if finalStr is not null then
           finalStr := finalStr || charString;
        else
           finalStr := charString;
        end if;
     end loop;

-- for inserting the XML data into the table
  rowsp := xmlgen.insertXML('scott.zipcodes',finalStr);

 dbms_output.put_line('INSERT DONE '||TO_CHAR(rowsp));
 xmlgen.resetOptions;
end;
/
```

Listing 9: Code for reading and XML documenting and inserting the data into the zipcodes table.

```
<?xml version = '1.0'?>
<Zipcodes>
   <mappings Record="4">
      <STATE_ABBREVIATION>CA</STATE_ABBREVIATION>
      <ZIPCODE>94301</ZIPCODE>
      <CITY>Palo Alto</CITY>
   </mappings>
   <mappings Record="5">
      <STATE_ABBREVIATION>CO</STATE_ABBREVIATION>
      <ZIPCODE>80323</ZIPCODE>
      <ZIP_CODE_EXTN>9277</ZIP_CODE_EXTN>
      <CITY>Boulder</CITY>
   </mappings>
</Zipcodes>
```

Listing 10: The XML document that is read by the code in Listing 9

```
SQL> select * from zipcodes;

ST ZIPCO ZIP_ CITY
-- ----- ---- --------------------
CA 95123 6111 San Jose
CA 95234      Sunnyvale
AK 72701      Fayetteville
CA 94301      Palo Alto
CO 80323 9277 Boulder
```

Listing 11: Output of select statement on the zipcodes table.

CONCLUSION

We have seen what Oracle tools are required to handle XML documents - to read into and out of the database. We used a simple table to see and understand the construction of DTDs. We used the DTD logic to define an XSL

document for the resultant XML. With the help of PL/SQL codes we saw how records from a table can be converted to XML documents and how data embedded in an XML document can be written back to the table. These basic models can be extended to work with complex tables with complex business rules. This will enable XML and the *database* to work closely together to facilitate the acquisition, integration, repurposing and exchange of data between enterprises.

ABOUT THE AUTHOR

Eashwar Iyer is a Project Manager for the Enterprise eCommerce practice at Quovera, in San Jose, California. He has over twelve years of experience in the computer industry. He has been involved in diverse projects in the US and India and has also conducted a training session on Oracle Designer at Oracle Corporation, Saudi Arabia. He has presented papers at IOUG-A Live!, Oracle Open World and NoCOUG chapter conferences. He holds a degree in Mathematics, a post-graduate degree in Management and a professional degree in Systems Analysis and Design.

Quovera provides strategy, systems integration and outsourced application management to Fortune 500, high-growth middle market and emerging market companies. The firm specializes in delivering intelligent solutions for complex enterprises, which improve productivity within the customer's business and optimize the customer's value chain, through integration of its customers and suppliers. The company also outsources the management of "best of breed" business applications on a recurring revenue basis. Quovera refers to its business model as "Intelligent – Application Integration and Management.
http://www.quovera.com