XML Support in the Database - XDB

Java in the Database - OracleJVM

AQ

Virtual Tables - Table Function

Query Result Formats

CONCLUSION
INTRODUCTION

Web services enable application interaction over the Web regardless of platform, language, or data formats. The key ingredients, including XML, SOAP, WSDL, and UDDI, have been adopted across the entire software industry. Web services usually refer to services implemented and deployed in middle-tier application servers. However, database vendors have started offering access to stored procedures as well as business data, through Web services interfaces. Database Web services represent a database-centric vision of Web services and work in two directions: accessing database resources as a Web service, and consuming external Web services from the database itself. Turning the Oracle database into a Web service provider leverages investment in PL/SQL packages. Conversely, consuming external Web services from the database itself, and integration with the SQL engine, enables Enterprise Information Integration.

This paper focuses on the advantages of opening up the Oracle9i Database, specifically PL/SQL packages, to the world of Web services, using the Oracle9i Application Server (Oracle9iAS) and the Oracle Database.

You will learn how you can easily publish PL/SQL packages as J2EE-compliant Web services to the Oracle9i Application Server. We also show you how to test and secure Web services, and how you can write static or dynamic Java clients to call them. We furnish additional background for using PL/SQL-specific legacy types and REF CURSORS, as well as how to adjust the mapping of SQL types in general.

If you need to call out from a database session to a Web service, we show how you can employ the SOAP client library in the Oracle9i Database to easily accomplish this in Java as well as in PL/SQL. Moreover, the integration of such call-outs with Table Function enables you to access results of Web service invocations as if they were database tables or views. We describe sample Web services call-out applications.

We also provide a general roadmap for the new world of Database Web services, mapping out Web services support for additional database capabilities, such as SQL queries, DML statements, XML operations, Advanced Queues operations, Streams operations, and Java Stored Procedures through synchronous as well as asynchronous invocation.

A BRIEF BACKGROUND ON WEB SERVICES

The World Wide Web Consortium (W3C) defines a Web service as a software application or component with the following properties.

- It is identified by a URI.
- Its interfaces and binding can be described in XML.
- It can interact directly with Web services through XML, by way of Internet-based protocols.
Because they are grounded in standard mechanisms and protocols, Web services are independent of implementation language (Java, Managed C++, JScript, Perl, VB.NET, C#, and J#), object model (EJB, COM, and so on), and platform (J2EE, .NET, and so on). In order to leverage Web services for your applications architecture, you must be able to encapsulate, describe, and publish existing applications as Web services and, conversely, be able to consume external Web services in your application. This paper describes both of these directions for database-centric applications written in PL/SQL or Java—how these applications can be made available as Web services to the outside and also how they, in turn, can utilize external Web services.

The terms SOAP, WSDL, UDDI, and RPC versus doc-invocation are central for understanding the Web services architecture. Here is a brief introduction of these terms before examining our database-centric use case.

- **SOAP** is an XML-based message protocol used by Web services. The transport mechanism (HTTP, FTP, SMTP, JMS, and so forth) is not prescribed. However, most Web services accept the firewall-friendly HTTP or HTTPS format.

- **Web services Description Language (WSDL)** is an XML document format that specifies the operations and their parameters—including parameter types—provided by a Web service. In addition, it describes the location, the transport protocol, and the invocation style for the service.

- **Universal Description, Discovery, and Integration (UDDI)** functions similar to a telephone directory—except that it lists Web services. It is a standard protocol that permits the registration of information about Web services, such as unique identifiers (white pages), business categories (yellow pages), and how to bind to a service protocol (green pages).

- **Web services come in many flavors.** Dispatching can happen in a synchronous (usually) or asynchronous manner, invocation can be performed in RPC-style (a single operation with arguments is sent and a response returned) or in message style (a one-way SOAP document exchange), and different encoding rules can be used (literal or encoded). When calling a Web service, you may know everything about it beforehand (static invocation), or you can discover its operations and transport endpoints on the go (dynamic invocation).

The following picture illustrates the different Web services components and how they interact.
Because Web service invocations can be seen as an exchange of XML messages, you could access and provide Web services at the transport level. However, it is not advisable to do this—you would have to fully understand the SOAP format, including the SOAP envelope, the SOAP header and its components, the encoding format used in the body, and so on. And you would have to parse and interpret all kinds of WSDL descriptions, as well as understand how to talk with a UDDI server.

Rather than writing and maintaining your own SOAP library and tools, you should use an existing and standard one. For Web service call-ins to the database, you can employ the J2EE Web services environment provided by Oracle9iAS. For calling external Web services from the database, you can also just reuse an existing SOAP client library. This paper describes both of these directions—first, Web service call-ins to the database, then calling external Web services from the database.

LEVERAGING THE DATABASE AS WEB SERVICE PROVIDER

Advantages of Database Web services Call-ins

Triggering database operations by Web services requests furnishes a standard, secure, and controlled way of opening up the database and sharing data, data logic, metadata, business entities, in a heterogeneous, non-connected corporate intranets. Database Web services call-ins allow:
- Leveraging existing or future PL/SQL stored procedures and Java classes running in the database.
- XML document storage and retrieval through SQL as well as XQuery, as Web services— for example, desktop applications can download, edit, and upload XML documents; applications can send and retrieve XML documents representing audit and logging information to the database for storage.
- Exposing Relational, Object, Text, Spatial, MultiMedia data storage and retrieval through pre-defined SQL queries and DML as well as XQuery operations, and so on, as Web services— for example, business intelligence services, catalog search services, and Map/GIS services.
- Exposing database queuing and messaging operations as Web services endpoints.

**Architecture**

The Oracle9i Database offers many resources and capabilities, ranging from SQL to PL/SQL and Java stored procedures, to the XML capabilities of the XML Database (XDB), to Advanced Queuing (AQ) and Streams. Java clients typically access these features through the Oracle JDBC driver and the Java Messaging Service (JMS). Oracle aims at going one step further and expose these capabilities through Web services. As a basis, we use Oracle9iAS, which provides a standard, scalable J2EE container together with a full J2EE Web Service Framework.

For developing, deploying and managing database Web services, Oracle provides the same complete and integrated tool set used for J2EE applications, including Oracle9i JDeveloper, Enterprise Manager and Web Services Assembler.
Publishing Database Web Services

Oracle9iAS enables you, currently, to create Web services based on J2EE components, as well as on Java classes and PL/SQL stored procedures. This section focuses on how you can publish a PL/SQL package, and furnishes details handled by our tools transparently for the developer. Assume that you have a package, MYAPP, available in the SCOTT schema. This discussion employs the Web services Assembler tool, available in Oracle9iAS version 9.0.3 and later, for creating Web services as J2EE-compliant EAR files deployed in Oracle9iAS Containers for J2EE (OC4J). Similar capabilities exist in earlier releases through an ant-based tool.

A Configuration File for Web Service Assembler

The input to the Web Service Assembler is provided through a configuration file, config.xml. This looks as follows for a Web service based on MYAPP.

```xml
<web-service>
  <display-name>Web Service using MYAPP</display-name>
  <description>Exposing the PL/SQL package MYAPP as a Web Service under the endpoint: /oow/MyApp</description>
  <destination-path>./myapp.ear</destination-path>
  <temporary-directory>./tmp</temporary-directory>
  <context>/oow</context>
  <!-- A stateless service based on a PL/SQL package. -->
  <stateless-stored-procedure-java-service>
    <!-- The URL under the context /oow for the service -->
    <uri>/MyApp</uri>
    <!-- Info needed at publishing time -->
    <jar-generation>
      <!-- Connection information -->
      <schema>scott/tiger</schema>
      <db-url>jdbc:oracle:thin:@localhost:1521:orcl</db-url>
      <!-- PL/SQL package information. By default, this is also used for the Java class name -->
      <db-pkg-name>MyApp</db-pkg-name>
      <!-- Java package information -->
      <prefix>mypack</prefix>
    </jar-generation>
    <!-- Info needed at runtime - the JNDI DB connection -->
    <database-JNDI-name>jdbc/OracleDS</database-JNDI-name>
  </stateless-stored-procedure-java-service>
</web-service>
```
Note that the connection information from <schema> and <db-url> is used only at code generation time. When you invoke the deployed Web service in Oracle9iAS, the JNDI data source jdbc/OracleDS specified in <database-JNDI-name> is used for connecting to the database. Therefore, make sure that this data source is defined and can access the MYAPP package.

**Deploying and Testing Your Web Service**

Use this configuration file to run the Web services Assembler tool and create a deployable EAR file, myapp.ear.

```
java  -jar WebServicesAssembler.jar  -config config.xml
```

After deploying the generated myapp.ear file into OC4J with Enterprise Manager, or the DCM control tool (or in a standalone OC4J installation with the admin.jar tool), you are ready to test the service. Using a host and port setting in accord with your Oracle9iAS installation, connect your browser to:

```
http://localhost:8888/oow/MyApp
```

You will see a list of the operations of your Web service and be able to test operations individually through your browser. In addition, at the endpoint /oow/MyApp?WSDL, you can access the WSDL specification of your service, and at /oow/MyApp?proxy_jar, you can obtain Java client classes that can be incorporated into a client application to invoke the Web service.

For more background information, refer to the Oracle9iAS Web Services Developer Guide, chapter 5, "Developing and Deploying Stored Procedure Web Services".

**Using JDeveloper to Deploy PL/SQL Web Services**

Oracle9i JDeveloper version 9.0.3 and later automates all the steps necessary to publish your Web services. All you must do is navigate from Connections to the desired database schema and expand the Packages node. Right-click the PL/SQL package, and choose Publish as Web Service to launch the PL/SQL Web services Wizard, which guides you through the steps for publishing your Web service. See also http://otn.oracle.com/tech/webservices/htdocs/series/plsql/content.html for a tutorial on how to use JDeveloper to publish PL/SQL Web services.

**Web Service Security and UDDI Publishing with Oracle Enterprise Manager**

With Oracle Enterprise Manager, you can secure Web services in the same way as all other J2EE components that run under OC4J. You can employ standards-based encryption, authentication, and authorization with PL/SQL Web services, use Single Sign-On, and centrally manage all aspects of security.

Oracle Enterprise Manager also enables you to publish your Web services to your Oracle UDDI Registry instance. When using the Deploy Application Wizard for deploying your Web service EAR files to OC4J, you can select the services...
(servlets) to publish to UDDI, provide the required classification information, and then add more details through the UDDI Registry Web Services Details window.

**Type Support and Mapping Flexibility**

If the PL/SQL-based database Web service created with the Web Services Assembler tool or with JDeveloper fits your needs, then you are done. However, you may encounter one or more limitations when using Oracle9iAS release 9.0.3: support is lacking for LOB types, XMLTYPE, REF CURSORs, and OUT as well as IN OUT arguments. Although all these limitations will be addressed in future releases, the following sections describe an alternative approach to obtain full flexibility in how PL/SQL methods and SQL types can be exposed as Web services. In order to do that, we will have to look at what is going on when you create a Web service based on a PL/SQL package.

**Oracle JPublisher: Mapping PL/SQL To Java**

Oracle JPublisher obtains the PL/SQL package specification from the database and generates a Java class (actually, SQLJ source code) for invoking the package's methods. To run JPublisher, type `jpub` on the command line. In our example we could have issued the following commands directly to obtain these classes.

```bash
jpub -sql=MYAPP:MyApp -package=mypack -user=scott/tiger \
   -url=jdbc:oracle:thin:@localhost:1521:orcl
sqlj -d $HOME/classes MyApp.sqlj
```

The `-sql` and `-package` options tell JPublisher that the MYAPP package should be published as a Java class, `mypack.MyApp`. Do you still recall the `<schema>` and `<db-url>` tags from our configuration file? Here you see how they are used in the JPublisher options `-user` and `-url`. The `sqlj` command line then translates the generated SQLJ file and places the compiled CLASS files under `$HOME/classes`.

The following section explains how to control the initial step of JPublisher code generation.

**User-Subclassing of Generated Code to Support REF CURSOR Arguments**

If you want to modify the behavior of generated code, have JPublisher generate code that will utilize a user-provided subclass. In our example, change the `-sql` option of JPublisher to read as follows.

```bash
jpub -sql=MYAPP:MyAppBase:MyApp ...
```

Now JPublisher places its code into `MyAppBase.sqlj` and also creates an initial version of `MyApp.sqlj`, where you can place your own code to override existing methods in `MyAppBase`. This is useful, for example, if you have a PL/SQL method that returns a REF CURSOR. JPublisher automatically maps the return type to `java.sql.ResultSet`. However, this `ResultSet` type cannot be published as a Web service parameter. You now have the following method on the `MyAppBase` class:
public ResultSet getRefCursor(String arg1, Integer arg2)

In MyApp, place a new method that can return the result set in a format supported by Web services, such as for example:

public String[] readRefCursorArray(String arg1, Integer arg2)
{
    java.sql.ResultSet rs = getRefCursor(arg1, arg2);
    ... create a String[] from rs and return it ... }

Now you must ensure that the getRefCursor method is omitted from the Web service operations and that the readRefCursorArray method is included instead. To accomplish this, define an interface, say MyAppInterf, that contains exactly those methods that you want to publish. Note that you can employ JPublisher to make this task simpler by using the following -sql option setting:

class-name=mypack.MyApp</class-name>— the name of the class that implements the Web service
•  <interface-name>mypack.MyAppInterf</interface-name>— the name of the interface that specifies the methods to be exposed
•  <java-resource>myapp_pregenerated.jar</java-resource>— the JAR file that we have just put together

Next, follow the same steps as before to deploy and test the Web service.

Use this approach on a per-method basis to map unsupported arguments into ones that are understood by Web services, or to map both the return and OUT (or IN OUT) arguments into a combined JavaBean structure returned by a method. However, if you can express your requirements as a mapping from a PL/SQL (or SQL) type to another SQL type that can be accomplished through a PL/SQL function in the database, then you can leverage the type mapping capability of JPublisher for your argument conversions.
Customizing Code With JPublisher Using SQL Conversion Functions

As an example, take the use of PL/SQL BOOLEAN arguments. This type is not even supported by JDBC. How can you expose a method with such arguments as a Web service? You must map BOOLEAN to another SQL type that is understood by JDBC as well as—in its Java format—by Web services, such as INTEGER. You must provide two conversion functions in PL/SQL: one that maps BOOLEAN to INTEGER, and one that does the opposite.

FUNCTION INT2BOOL(i INTEGER) RETURN BOOLEAN;
FUNCTION BOOL2INT(b BOOLEAN) RETURN INTEGER;

Now we just need to tell JPublisher that the SQL BOOLEAN type is mapped to a Java Boolean in the generated code. In reality, we expect to send and receive from the database the SQL INTEGER type instead (this works for JDBC because Java Boolean can receive a numeric SQL value). Moreover, in the server we perform conversions between these two SQL types with the INT2BOOL and BOOL2INT functions. Specify this with the following command-line setting.

jpub -addtypemap=BOOLEAN:boolean:INTEGER:INT2BOOL:BOOL2INT ...

This particular conversion is actually already preprogrammed in JPublisher (just make sure that the SQL file [Oracle Home]/sqlj/lib/sqljutl.sql has been installed in the database if the conversion functions are not found). However, the illustration should convey the concept of what you can achieve through such type conversions. Also note that if a convertible type occurs as an OUT or IN OUT argument of a stored procedure or function, then JPublisher generates additional PL/SQL wrapper code into a SQL file that you must load into the database before you can use the generated code.

Both approaches help you obtain the desired behavior for your database Web service call-ins. You may also have already noticed that through the capability of pregenerated—meaning user-written—JAR files, you are able to realize any database Web service, as long as all that it requires is a database connection at runtime. All you need to do is write a Java class that can be instantiated with a Connection object.

Now that we know how to create Web services that make calls into the database, we will turn our attention to code actually running inside the database as part of a stored procedure and that would like to call out to a Web service. How can this be accomplished?

FEDERATED INFORMATION—CONSUMING EXTERNAL WEB SERVICES IN THE DATABASE

The previous section described how to make database resources available and accessible, through Web services. This section discusses the advantages and instructions for turning external Web services into SQL data sources.
**Advantages of Database Web services Call-outs**

Beyond the benefits of a single, fast and scalable Relational, XML, Text, and MultiMedia data repository, there is an increasing need to combine data from multiple sources, including web services, and be able to query the resulting data as if it were in one single, federated database. There are multiple use cases for combining the power of SQL with a federated view of heterogeneous data. For example, tracking and mining stock prices, refreshing and mining scientific data (such as genome data), querying and refreshing IRS tax tables, and tracking and mining weather information (see our sample application demo) are the building blocks for the emerging Enterprise Information Integration.

In addition, data can trigger external Web services invocation. You can leverage business information such as inventory status, orders status, and so on, that are tracked within database tables, for status-driven computation. Some examples are automated order tracking and processing, shipping tracking, and Web shopping processing (see our sample application demo).

**Architecture**

In order to consume external Web services, we must be able to call out to these services. This section furnishes an overview of extending database functionality, as well as integrating data from Web services with the SQL engine.

As mentioned earlier, at its most basic level, you could invoke a Web service directly over an HTTP connection. For an example of this technique that uses HTTP call-outs from PL/SQL, go to [http://otn.oracle.com/tech/webservices/htdocs/samples/dbwebservice/DBWebServices_PLSQL.html](http://otn.oracle.com/tech/webservices/htdocs/samples/dbwebservice/DBWebServices_PLSQL.html). This approach may be acceptable if you know the Web service and its format ahead of time, and if you are willing to read and interpret the service's WSDL specification yourself. A more generic solution is to use an existing SOAP stack that understands WSDL, permitting you to dynamically interact with a Web service and utilize pre-generated client-proxy code. A client-proxy simplifies Web service invocations: once you have created a proxy instance, you can call the desired Web service operations on it.
Installing a SOAP Client Stack in the Oracle Database

All we need to do now is to get a Java client stack for SOAP and install it in the OracleJVM to make it available to database Java code. The following instructions have been successfully tested against Oracle9i Database Release 2, which embeds a J2SE 1.3.x compatible OracleJVM; these instructions might work with previous releases of the database that embed a J2SE 1.2.x compatible OracleJVM as well.

If you want to use the current Oracle SOAP classes (available in OC4J download), issue the following command:

```
loadjava -thin -user sys/<sys-passwd>@<host>:<port>:<SID> -resolve -synonym -verbose -grant public
[OC4J_HOME]/soap/lib/soap.jar [OC4J_HOME]/lib/dms.jar
[OC4J_HOME]/jlib/javax-ssl-1_1.jar
[ORACLE_HOME]/lib/servlet.jar
[OC4J_HOME]/jdk/jre/lib/ext/mail.jar
```

Note that we are loading the classes into the SYS schema. By using the -synonym and -grant public options, we are also making them available as well as executable from other schemas. In addition, to every user who will be making call-outs, we must grant permission to use networking sockets. If you specify ‘*’ for host and port, you are giving unrestricted access for connections.

```
execute dbms_java.grant_permission
('<db-user>','SYS:java.net.SocketPermission',
'host':<port>','connect,resolve');
```

Instead of Oracle SOAP, you might want to use JAX-RPC client. In this case, load the following JAR files from the JAX-RPC reference implementation instead:

```
activation, commons-logging, dom, dom4j, jaxp-api, jaxrpc-api, jaxrpc-ri, mail, saaj-api, saaj-ri, and sax.
```

The beauty of an integrated Java VM in the Oracle database is the ability to extend database functionality simply by loading Java-based libraries such as the SOAP client. Furthermore, in order to keep up with the fast pace of Web services infrastructure, the SOAP stack can be updated simply by loading a newer version.

Static and Dynamic Web Services Clients in Java

If you want to call a Web service from server-side Java code, you are already done. Either you program against the dynamic invocation interface of your Web service client stack, or you obtain, load, and use static client-proxy JAR files. In both cases, you can employ (and deploy) the same Java code that you would write for a standalone Java client with the same SOAP library. Remember that you can use JDeveloper to generate client proxies for Oracle SOAP, and for JAX-RPC you can utilize its wscompile tool. Now that we have Java clients, what about PL/SQL?

Web Services Clients in PL/SQL

We know that we can publish static Java methods as PL/SQL stored procedures and functions. However, Web service operations are provided as instance methods on the client-proxy class. This means that we have to provide some simple Java glue code to expose Web service operations as static methods on a Java class. Only
then can we publish the operations with a PL/SQL call specification. The end result is that we can invoke the Web service just like any other standard PL/SQL procedure from PL/SQL code as well as from the SQL top level. Although you must go through these manual transformations right now, these tasks will, no doubt, soon become automated.

**Using Table Functions to Integrate Web Services with the SQL Engine**

Taking a data-centric perspective, we may be interested not just in individual Web service call-outs, but actually in materializing over a range of input values the graph of a Web service function as a SQL table or view structure. This enables us to fully integrate Web service information from heterogeneous sources through the heart of the database – the SQL engine.

To accomplish this, we enlist table functions that can materialize table rows over a range of input values. Table function results are typically pipelined – the returned rows are streamed directly to the consuming process without intermediate staging. If the underlying Web service function is deterministic, this is also exploited in the table function.

The following example shows how to create a table function around the Web service exposed through PL/SQL as the following function

```sql
FUNCTION CALL_WS(s VARCHAR2) RETURN VARCHAR2
```

We must define a SQL type for holding the graph of the Web service and also the table function itself that returns these table rows in pipelined fashion.

```sql
CREATE TYPE WS_ROW_T -- hold graph of Web Service function
    AS OBJECT (res VARCHAR2(255), s VARCHAR2(255));
CREATE FUNCTION WS_TABFUN(cur SYS_REFCURSOR) -- table function
    RETURN WS_ROW PIPELINED AS
    s VARCHAR2(255);
BEGIN LOOP FETCH cur INTO s;
    EXIT WHEN cur%NOTFOUND;
    PIPE ROW(WS_ROW_T(s,CALL_WS(s)));
END LOOP;
END WS_TABFUN;
```

Now we can use the table function as a virtual table. In the code below, the inner `SELECT` creates rows whose columns are used as arguments for invoking the `CALL_WS` Web service call-out. The table expression could be used in other SQL queries, for constructing views, and so on.

```sql
SELECT <some-columns> FROM
    TABLE(WS_TABFUN(CURSOR(SELECT s FROM <some_table>))),
    ... WHERE ...
```
Web services Call-Outs Demos -- Sample Applications

The following demos illustrate consuming external Web services from within an Oracle database using direct call as well as the use of table functions. For the complete applications source code, upcoming demos, and "How To", check the Web services center page on the Oracle Technology Network at http://otn.oracle.com/tech/webservices/content.html.

Processing Web Shopping Orders

This sample implements a purchasing order processing scenario.

(i) The user can browse through the products catalog and purchase, on-line.

(ii) Based on the login information supplied by the user, a Credit Agency Web service --- which maintains a repository of customers and their credit authorization --- authenticates and authorizes the user for buying products of interest. The status of the order will be stored as 'PENDING' and processed and shipped on a later time.

(iii) A database batch job scheduled to run every six minutes (for demo purposes), processes "PENDING" orders by calling-out the Credit Agency Web service with the customer information, the authorization id (obtained previously), and the exact amount of shipment (which might be different from the initial order amount). If the response to payment request to the Credit Agency Web service is okay, then the order status is changed from "IN_SHIPPING", to "SHIPPED". This marks the end of the life cycle for this particular order. The user can view the status of the order at any time.

Tracking and Mining Weather Temperature

This sample implements a weather reporting system that can be used by the end users to mine temperature reports of some US cities.

(i) The application tracks the temperature of important cities in the US, available through a public web service at http://services.xmethods.com/ve2/ViewListing.poj;jsessionid=hnXezdkbZATXgSUvQtxPuM(QhxicSRM)?serviceid=8.

(ii) A table function that takes a range of input values (zipcodes) as arguments is used to provide the resulting data of external web service invocation, as a virtual relational table.

(iii) Mining Current temperature: the end user can request the current temperatures of selected cities -- which will result in directly invoking the Temperature Web service with the zipcodes of the selected cities -- and apply the full power of SQL including aggregate functions such as avg, min and max on the resulting data.

(iv) Mining Recently Collected Temperature: a database table is used to store temperatures collected since a determined period of time. A database batch job is scheduled to run every six minutes (for demo purposes) and calls-out the
Temperature Web service with the zipcodes of all the cities and store the data in a real database table. The end user can "mine" the recently collected temperatures of selected cities, from this table, using the full power of SQL including aggregate functions such as avg, min and max.

This tracking and mining approach can be applied to any "business data" available as external web services such as stock prices, scientific data, IRS tax tables, and so on.

DATABASE WEB SERVICES - A ROADMAP

Let us take a step back and reflect on, or rather generalize, the examples and illustrations provided here. Oracle Database Web services will evolve around the following two capabilities:

- Database Web services call-ins: being able to invoke services implemented using all database APIs, including SQL, XML, PL/SQL, Java, Advanced Queuing, and Streams, through standard Web services mechanisms.
- Database Web services call-outs: being able from the database itself to invoke external Web services, such as those based on J2EE and .NET, and combine with the power of SQL.

Oracle Database Web Services Building Blocks

We provide hereafter, a non-exhaustive list of enabling technologies that we will be leveraging in the future.

Java/J2EE Web Services Framework

For database as service provider (Web service calls to the database), we will leverage existing and upcoming Web services capabilities of Oracle9iAS, built on top of the Oracle9iAS J2EE stack. Therefore, interoperability, J2EE deployment artifacts as well as the built-in quality of service, such as: security, scalability, reliability, availability, and performance, all come free of charge. Similarly, for database as service consumer (Web services call-out), we will leverage the client subset of Oracle9iAS Web services framework to accommodate static as well as dynamic service clients directly in the database.

SOAP with Attachment

Database Web services will leverage SOAP with Attachments when returning large XML documents and binary data such as BLOBs and CLOBs -- in a separate mime part -- as a better alternative to in-lining the data within the SOAP envelope.

PL/SQL

PL/SQL is the Oracle procedural extension to SQL; it provides a tight and seamless integration with SQL and database types. PL/SQL is used extensively by a large community of Oracle developers for implementing data centric logic and metadata. PL/SQL stored procedures are the first-class citizens of database Web
As highlighted above, raw PL/SQL can be used for composing and issuing the SOAP request message, but we found the Java approach (exposable to SQL and PL/SQL through wrapper), much simpler.

**JDBC - Web RowSet (JSR 114)**

The JDBC RowSet specifies the ability to create a RowSet (tabular data) from a datasource, usually a database. CachedRowSet enables disconnected RowSet such that the row data and metadata can persist beyond the connection, with the ability to propagate and synchronize any modifications to the rowset back to the database. The JSR 114 specification defined WebRowSet as CachedRowSet serializable to and from XML. WebRowSet can easily be shared across J2EE components, as well as Web services. Database Web services will leverage this standard capability as one option for returning query results.

**XML Support in the Database - XDB**

XML documents are at the heart of Web services. Oracle9i Release 2 furnishes a native XML support (XDB) that enables scalable XML document storage and retrieval, as well as seamless manipulation of SQL data and XML, using both SQL operations and emerging XML standards. Database Web services will leverage all existing and upcoming XML capabilities in the database, such as XMLType, XML Schema, XPath, SQLX, the XML Developer's Kit (including the XML SQL utility known as XSU), and XQuery.

**Java in the Database - OracleJVM**

Since Oracle8i, Release 1 (Oracle 8.1.5), Oracle has offered a tightly integrated JVM that supports Oracle's database session architecture. Any database session can activate a virtually dedicated JVM during the first Java code invocation; subsequent users then benefit from this already Java-enabled session. In reality, all sessions share the same JVM code and statics—only private states are kept and garbage collected in an individual session space, to provide Java sessions the same session isolation and data integrity capabilities as SQL operations. This session-based architecture provides a small memory footprint, and gives OracleJVM the same linear SMP scalability as the Oracle database. There is no need for a separate process for data integrity. OracleJVM enables the extension of database functionality, by loading and resolving standard Java libraries and making them available to all database sessions (using JAX-RPC client for database as service consumer). The Java class representing the Web service client-proxy is deployed within the JVM in the database, and performs the calls to the service methods, on behalf of the database operation consuming the external Web services.

For database as service provider, OracleJVM enables Java classes deployed in the database to be invoked either through JDBC and the PL/SQL wrapper, or directly through an upcoming, RMI-like, database's Java native interface.
**Advanced Queuing (AQ)**

Advanced Queuing (AQ) is Oracle database's integrated message queuing system, which brings to messaging all the inherent benefits of a database, such as reliability, integrity, high availability, security, and scalability. For database Web services, we will expose database queue operations such as enqueuing and dequeuing messages, as well as queue administration operations. We will also leverage AQ and an execution agent such as DBMS_JOB, to allow deferred, asynchronous invocation of database operations (SQL statement, PL/SQL packages, and Java classes).

**Virtual Tables - Table Function**

Table functions\(^1\) are functions written in PL/SQL, Java, or C that produce a collection of rows that can be queried like a physical database table, by using this virtual table in the FROM clause of a SQL query. Table function can take a set of rows or cursor as input parameter. Database Web services leverage this functionality and its future enhancements, by applying SQL operations (for example, on data resulting from external Web services calls).

Oracle9i Database improves table function query performance and memory consumption through streaming, pipelining, and parallel execution:

- **Parallel execution**: enabling multithreaded, concurrent execution of table functions
- **Streaming**: eliminating intermediate staging between processes
- **Pipelining**: iteratively providing result rows from the collection returned by a table function as the rows are produced, instead of waiting until the entire collection is staged in tables or memory and then returning the entire collection

**Query Result Formats**

In the first phase, Database Web services will materialize the query result in the following ways:

- **As an array of a JavaBean**, where the JavaBean represents one row of the result set
- **As an XML document**, using the format and description of Oracle's XML SQL Utility
- **As an XML document**, using the WebRowset format and description

Specific XML schemas are currently being defined by industries or consortia. Over time, we will investigate support for these schemas as well.

---

CONCLUSION

We have outlined how you can leverage database-centric Web service capabilities, using Oracle9i Application Server and Oracle9i Database. These products will enable your database to act both as a consumer and a provider, in a global or corporate-level Web services architecture. Database Web services leverage your existing server-side infrastructure built on SQL, PL/SQL, and Java. Beyond that, Database Web services will use additional capabilities of the Oracle database, such as native XML document storage and processing through both SQL and XML standards, the Advanced Queuing capability for messaging and deferred execution.

The Database Web service approach, when taken to its ultimate conclusion, will virtualize and globalize database schemas, tables, queues, and their operations.

Oracle is committed to helping you realize the full potential of the combined power of the Oracle Database and Application Server through Database Web services. You are invited to check back for updates to this exciting new technology in the months to come, at the Oracle Technology Network at http://otn-stage.us.oracle.com/tech/webservices/database.html, and provide feedback through the Web services discussion forum at http://www.oracle.com/forums/forum.jsp?id=642407.