WEB SERVICES IN ORACLE DATABASE 10G AND BEYOND

Ekkehard, Robwedder, Oracle Corporation
Kuassi Mensah, Oracle Corporation

INTRODUCTION

The dream of Web Services is to allow application interaction over the Web irrespective of the provider’s or consumer’s platform or language. Moreover, Web Services are the foundational technology for Grid Services – a set of emerging technologies and standards for optimally managing Enterprise resources and services. This paper describes how the Oracle Database 10g is integrated into the Web Services stack and can be leveraged as a framework for heterogeneous data sources, as well as a provider of application modules built with SQL, PL/SQL, Java, or other database resources. Additionally, we outline how Database Web Services will be evolving into Grid-specific Data Access and Integration Services.

The Oracle Database 10g can act both as a Web Service consumer and as a Web Service provider. In the former case you can call out from database queries, triggers and applications, whether written in PL/SQL or in Java, to external Web Services. In the latter case, database resources such as PL/SQL packages, Java stored procedures, SQL queries, DML statements, and XML data are published through the Oracle Application Server as Web Services. In essence, this paper provides an update to the documents [1,2] based on the features provided in the Oracle Database 10g and Oracle Application Server 10g products.

Two tools come into play in this endeavor. The Web Services Assembler command line utility is able to generate Database Web Services for the Oracle Application Server. It transparently utilizes the JPublisher tool, which is employed to generate Java wrappers for database resources, such as PL/SQL packages, queries, SQL DML statements, or Java stored procedures. Additionally, you can use JPublisher directly in order to implement call-outs to external Web Services from the database. If you just need to expose PL/SQL code as Web Services you may want to choose the simplest road to creating Database Web Services – the 9i JDeveloper integrated development environment, which uses JPublisher and Web Services Assembler transparently through its Web Services wizards.

This paper is structured as follows. First, we provide a brief introduction to the main concepts of Web Services and Grid Services. Then we consider in detail how external Web Services can be invoked from inside the Oracle Database. Next we describe the mechanism of publishing database resources as Database Web Services hosted by the Oracle Application Server. Finally we consider the future evolution of Database Web Services and its relation to Grid Services.

INTRODUCTION TO WEB SERVICES AND GRID SERVICES

WEB SERVICES CONCEPTS

Web Services are software applications or components identified by a URI that provide a description of their interfaces and binding in XML, and that can interact directly with other Web Services through XML via Internet-based protocols.

Web Services' use of standard formats and protocols provides independence from a particular implementation language (Java, Managed C++, JScript, Perl, VB.NET, C#, J#), object model (EJB, COM, and so on), and platform (J2EE, .NET, and so on). At its core, Web Services transmit information represented in XML format over various transports, such as HTTP, FTP, SMTP, JMS, etc. Currently, most Web Services accept the firewall-friendly HTTP or HTTPS format. The XML itself is formatted according to the XML Schema Definition [3]. The SOAP messaging protocol [4] defines the general shape as well as processing rules for messages that flow between Web Services. For a given Web Services endpoint (or port), the format of messages that are transmitted to it and with which it may replies are set down in its Web Services...
Description Language (WSDL [5]) document. Specifically, this includes the permitted operations with parameter and return types and instructions on how to represent programming language bindings in XML format. Typically, you utilize the WSDL for a particular Web Service in order to interact with that service, either via a dynamic invocation capability in your programming platform, or via a generated static client proxy for your programming language. When creating a Web Service based on a programming language artifact, such as a Java class or a PL/SQL package, the WSDL will typically be generated for you. Finally, the UDDI standard [6] provides a standardized registry facility for locating Web Services and their descriptions. At one level you can treat Web Services as a specific flavor of XML messaging. However, given the complexity and the evolving nature of the Web Services practices and standards, you will typically rely on a Web Services technology stack and rely on tools to publish Web Services and their clients. This allows you to minimize changes to your own code while maximizing interoperability with other Web Services.

**MOVING ON TO GRID**

Grid is about efficient utilization of computing resources (software, hardware, application, and so on). The Global Grid Forum (GGF) has defined a Grid architecture based on an enhanced Web Services technology stack (Open Grid Services Interface, OGSI [7]). Of particular interest for database and Database Web Services users are the specifications of the Grid Database Access and Integration Services working group (DAIS-WG). They describe how to access databases as Grid resources, and how to integrate database results over a Grid topology. We will show how the DAIS effort constitutes a superset of our current Database Web Services functionality.

**TO SERVE, OR TO BE SERVED…**

As mentioned earlier, there are two ways in which the database can be involved in Web Services: either by serving up database resources as a Web Service from the Oracle Application Server, or by calling out to external Web Services from the Oracle Database. We describe both of these directions in this paper. First we turn to Web Service call-outs from the database.

**DATABASE AS WEB SERVICE CONSUMER**

In order for code running in the database to access other Web Services, such as stock quotes, weather information, Web search results, scientific data, or enterprise data available through Web Services we need to be able to call out to these services. We can do this by utilizing a JAX-RPC [8] based Web Service client in the database,

**EMPLOYING A JAX-RPC WEB SERVICES CLIENT STACK WITH THE ORACLE DATABASE**

In Oracle Database 10g, the libraries sqljutl.jar and utl_dbws_jserver.jar from [Oracle Home]/sqlj/lib, as well as the SYS.UTL_DBWS package come preloaded into the database at install time. You can also load these JAR files manually, for example if you are using a pre-10g database such as Oracle 9i Rel 2, or if you want to install an updated version of the client. In this case the SYSDBA can run the script [Oracle Home]/sqlj/lib/initdbws.sql, as well as the scripts utl_dbws_dec1.sql and utl_dbws_body.sql in the sqlj/lib directory.

While users may also be able to substitute the Apache Soap stack instead of JAX-RPC –see [1,9] for more information– we are discussing Web Services callouts based on a 10g JAX-RPC here. This solution also provides the following functionalities not available with other Web Services client stacks: the capability to natively invoke server-side Java methods; a PL/SQL package UTL_DBWS for dynamically invoking Web Services from PL/SQL; the automatic generation of static client proxy code for both, Java and PL/SQL; and automatic upload and installation of the generated Java and PL/SQL code into the database. Irrespective of which solution you are using, you will have to grant database users permission to connect outward from the database to external sockets.
**STATIC AND DYNAMIC CLIENTS**

In general, you have two choices for setting up Web Services clients. In both cases you utilize the WSDL description of the service itself. In the **dynamic client** you set up service and call objects at runtime and you invoke an operation through a generic list of input arguments (of type Object in Java, or ANYDATA in PL/SQL), which will produce a return value as well as a list of output values. For **static client** invocations the WSDL document is utilized to generate appropriate Java proxy classes or PL/SQL packages for the described Web Services ports. The following diagram illustrates the various flavors of Web Services clients in the database.

![Figure 1 - Database Web Service Call-Outs based on the JAX-RPC stack.](image)

**WEB SERVICES CLIENTS IN JAVA**

If you are using Java, you may want to program with the JAX-RPC Dynamic Invocation Interface (DII) API directly. In this case you just have to place your code into the server-side JavaVM and are done. Alternatively, you may prefer to invoke the external Web Service through statically generated client proxy code. JPublisher [10] fully automates the tasks of generating the client proxy code, compiling it, jarring it up, and loading it into the database. For example, it is sufficient to provide the following command line.

```
jpub -proxywsdl=URL_of_Web_Service_WSDL -user=username/password
```

Additionally, you can specify `-endpoint=external_Web_Services_URL` if this is different from what is provided in the WSDL itself, and `-url=JDBC_database_URL` if different from the default database instance accessed with the OCI driver. JPublisher will generate a static client-proxy JAR based on the referenced WSDL document and automatically load the proxy classes into the database. To prevent JPublisher from loading the generated code into the database you can specify the option `-proxyopts=noload`. The advantage of using static client proxies is that you can simply reference port-type instance methods in your Java code without worrying about how to marshal or unmarshal individual arguments from the SOAP message.

**WEB SERVICES CLIENTS IN PL/SQL**

In Oracle 10g we are also providing the PL/SQL package UTL_DBWS for invoking Web Services dynamically from PL/SQL code. This package is itself implemented in terms of the JAX-RPC DII implementation. Given a WSDL document you can determine the qualified names of the services described in it. A function is provided to create an instance of a service `object` and return a handle to it. Subsequently, you can discover the ports on the service and return handles to call instances for these. If omitted, the service and port designations default to the first service and –respectively– the first port described in the WSDL. The call object can then be used to dynamically invoke the Web Service in a generic manner: input, output, and return values are represented using the SQL ANYDATA type. (At the time of this writing this package limits Web Service operations to parameters that can be mapped to simple SQL types.)

Analogous to static Java proxy support for Web Services we also provide support for static PL/SQL clients. In this case Web Service clients are mapped to PL/SQL packages with the functions and procedures corresponding to the various operations of the Web Service. Structured types are mapped to corresponding SQL object types.

In order to support static PL/SQL clients the JPublisher utility has to perform the same work as for the static Java clients and then some. In order to expose Java code in PL/SQL you need to provide static Java methods and not dynamic ones, such as are generated for standard Java proxies. Together with this additional Java glue code one also requires PL/SQL call specs for invoking Java as well as any additional SQL type definitions for representing more complex types. The good thing is this is all generated automatically and transparently by JPublisher. As a matter of fact, by default JPublisher will provide both Java and PL/SQL versions when asked to create a static Web Service client. The bottom line is that you can invoke external Web Services simply like any other standard PL/SQL procedure from PL/SQL code as well as from the SQL top
level. Note that several JPublisher front-end flags also affect the location and naming of the Java and SQL code generated by JPublisher: -dir, -package are the directory and respectively package name for generated Java code, while the -plsqlfile and -plsqlpackage options guide similar behavior for generated PL/SQL code.

**WEB SERVICES DATA SOURCES**

Once a Web Service operation is represented as a PL/SQL function, it only requires one small step to wrap a table function around it. Typically you would use this to materialize the graph of the Web Service invocation over a range of input values. Additionally, you can exploit determinism in the Web Service function itself. With this approach Web Services can become first-class SQL entities in the form of tables and views and permit you to fully combine information from heterogeneous sources, including external Web Services. A worked-out example with table functions can be found at [11].

**DATABASE AS WEB SERVICES PROVIDER**

The Oracle Database provides a number of resources and capabilities, ranging from—all of course—SQL to PL/SQL and Java stored procedures to the XML capabilities of the XML Database (XDB) to Advanced Queueing (AQ) and Streams. From Java clients this is typically accessed through the Oracle JDBC driver and the Java Messaging Service (JMS). In order to expose this functionality through Web Services we utilize the Oracle Application Server 10g, which provides a standard, scalable J2EE container together with a full J2EE Web Service Framework.

**Figure 2 - Using Oracle Application Server for Database Web Services Call-Ins.**

Database Web Services are deployed just like other J2EE applications, and you use the same tools and procedures as for any other J2EE component for creating, deploying, securing, and managing Web Services. This comprises the Oracle JDeveloper integrated development environment, Oracle Enterprise Manager, as well as command-line based tools.

**SQL AND PL/SQL TYPES SUPPORTED BY JPUBLISHER**

Before considering the various flavors of Database Web Services provided by Oracle Application Server 10g we need to take a closer look at JPublisher, our workhorse for supporting Database Web Services functionality. Specifically, we will examine the SQL and PL/SQL type support that it provides. Fundamentally, JPublisher maps database resources into Java classes. The methods on the class correspond to operations of the exposed Web Service. The Web Services Assembler tool from the Oracle Application Server then takes this class and publishes it as a Web Service. There are two differences between a JPublisher-generated Database Web Services class and any other Java class that is exposed as a Web Service. For one, the JPublisher-generated class must be constructed with a JDBC connection object, which—at runtime—will be initialized based on a JNDI datasource whose name is provided in the application. Secondly, Database Web Services are always published as stateless Web Services. The way in which JPublisher maps SQL and PL/SQL types affects whether an operation can be published and how its corresponding WSDL description looks like. With the Oracle Application Server 10g [9.0.4] and later the following SQL types can be published in Database Web Services signatures: simple SQL types including DATE, SQL object types, SQL VARRAY types, SYS.XMLTYPE, and REF CURSOR. The following PL/SQL-specific types are also supported: BOOLEAN, RECORD types, and index-by table types. Moreover, JPublisher supports the publishing of OUT and IN-OUT arguments. We will consider various aspects of these type mappings before surveying the different Database Web Services capabilities that are provided.

**SUPPORT FOR OUT AND IN-OUT ARGUMENTS**

In Oracle Application Server 10g the JAX-RPC based Web Services employ holder types to model OUT and IN-OUT parameters. This was not supported in 9.0.4. There, whenever a Web Service is returning parameters, JPublisher will create a Java bean that represents the return value—if any—as well as all of the OUT and IN-OUT parameters. The 9.0.4 Web Services Assembler can then publish the service with input parameters and this single return type.
**MAPPING SQL REF CURSOR**

REF CURSOR arguments are materialized in Java as `java.sql.ResultSet` parameters. This type is not directly mappable into a suitable XML type. Several choices are provided. If the shape of the result set can be discovered (such as, for example, if it is returned directly by a known SQL query) it can be mapped into an array of Java Beans, each element of which represents a row in the result set. In addition, result sets may also be mapped into more generic XML representations, such as into a WebRowSet as defined in JSR 114 [12], or –using the SQL/XML specification [13] or Oracle's XML SQL Utility (XSU [14])– by mapping into an XML fragment, such as represented with `org.w3c.dom.DocumentFragment` or `javax.xml.transform.Source`, for example. The specific code generation can be controlled by JPublisher flags, or through the class path environment in effect when the main JPublisher entry point `oracle.jpub.doit.main()` is invoked. Subsequently, depending on the Web Services stack that is employed, these types will be mapped into appropriate XML types in the WSDL document, such as `xsd:ANY`.

**SUPPORTING SYS.XMLTYPE**

By default JPublisher retrieves and sends SYS.XMLTYPE values through Oracle's `oracle.xdb.XMLType` representation. This in turn is mapped into `org.w3c.dom.DocumentFragment` in order to be published as parameters in Web Services. Unfortunately, XMLTYPE is only fully supported in the JDBC OCI driver. As a workaround JPublisher also provides the command line options `-style=webservices9` which will result in converting XMLTYPE values into strings while in the database server. These are subsequently materialized as `DocumentFragment` arguments in signatures exposed as Web Services.

**PROVIDING SUPPORT FOR PL/SQL-SPECIFIC TYPES**

Unlike other client programming APIs, the JPublisher tool provides support for PL/SQL-only types. It accomplishes this by converting PL/SQL-only types in the database itself into corresponding SQL types. This is achieved through server-side SQL type definitions, conversion function definitions, and –as required– PL/SQL wrapper code for individual functions and procedures. The following describes how various types are supported.

PL/SQL BOOLEAN is mapped into Java `boolean`. This requires that the script `[Oracle Home]/sqlj/lib/sqljutl.sql` be installed in the database where the JPublisher-generated code is to be run. (By default this is the case in 9.2 and later database installations that are Java-enabled.)

PL/SQL RECORD types as well as PL/SQL TABLE types result in JPublisher generating corresponding SQL object or –respectively– SQL VARRAY types together with conversion functions for mapping between SQL and PL/SQL types. On the Java side, instances of these types are received into JPublisher-generated wrapper classes. These relationships are described in type maps that JPublisher generates and that can be used as input to subsequent JPublisher runs. This makes it possible to incrementally build up and use complex type mapping scenarios. In addition, if you are using the JDBC OCI driver and require only the publishing of scalar indexed-by tables, you can also employ a direct mapping between Java and such types.

**USING WEB SERVICES ASSEMBLER FOR PUBLISHING DATABASE WEB SERVICES**

The Oracle Application Server permits you to create Web Services based on J2EE components, as well as on Java classes and on database resources, such as PL/SQL packages, SQL queries and DML statements, and Java stored procedures. Here we describe how all four of these database resources may be published as Web services. In this discussion we describe the Web Services Assembler tool in the 10g Oracle Application Server. This tool is used for creating Web Services as J2EE compliant EAR files that are deployed in OC4J. It transparently invokes the JPublisher utility for generating Java code corresponding to the respective Web Service.
A CONFIGURATION FILE FOR WEB SERVICE ASSEMBLER

The input to the Web Service Assembler is provided through a configuration file config.xml. For Oracle Application Server 10g, whenever you want to expose a database resource, you would need to provide information such as the following in the configuration file.

```xml
<db-port>
  <!-- The uri to be used within the servlet context -->
  <uri>/statelessSP</uri>

  <!-- The database schema for which to generate a Web Service -->
  <schema>scott/tiger</schema>

  <!-- The database connection string to generate a Web Service -->
  <db-conn>jdbc:oracle:thin:@localhost:5521:sqlj</db-conn>

  <!-- The datasource used during Web Service runtime -->
  <datasource-location>jdbc/OracleDS</datasource-location>

  <!-- The name of the generated Java interface (optional) -->
  <port-name>Company</port-name>

  <!-- The package name for the generated Java code (optional) -->
  <prefix>acme</prefix>

  <!-- Directives with JPublisher processing flags (optional) -->
  <jpub-input>
    jpub.omit_schema_names
    jpub.toString=true
  </jpub-input>

  ... Information about the database resource that is published as a Web Service ...
</db-port>
```

Once you have written the complete configuration file, you invoke Web Services Assembler to generate your service.

```
% java -jar ws.jar -config config.xml
```

PUBLISHING PL/SQL PACKAGES AS WEB SERVICES

For exposing a PL/SQL package as a Web Service we require the name of the database package to be published. Optionally we can provide a list of method names if only a subset of its functions and procedures should be included.

```xml
<plsql-package>
  <!-- The database package to be exposed. -->
  <name>Company</name>

  <!-- A list of methods to expose (optional). -->
  <method>method1</method>
  <method>method2</method>
</plsql-package>
```

While the Web Services Assembler tool transparently invokes JPublisher, you can also call JPublisher directly from the command line. This is useful if you simply want to generate a Java class that “wrappers” a given PL/SQL package and do not care about turning this class into a Web Service. You will recognize some pieces of our configuration file in the following JPublisher command line invocation (make sure to quote arguments as required by your command line shell!).

This generates the Java class `acme.Company` for calling the PL/SQL package SCOTT.COMPANY.

```
jpub -user=scott/tiger -url=jdbc:oracle:thin:@localhost:5521:sqlj -package=acme \
  -sql=Company(method1,method2) -omit_schema_names -toString=true
```
**PUBLISHING SQL QUERIES AND SQL DML STATEMENTS AS WEB SERVICES**

If you want to create a Web Service based on a SQL query or on an UPDATE, INSERT, or DELETE statement, you could place your SQL code inside of a PL/SQL package and then publish that. You can also accomplish the same goal more easily by directly specifying the SQL code in the configuration file. The following is an example of the kind of information that you would to provide for your database resources in this case.

```xml
<sql-statement>
  <operation>
    <!-- Name for the Web Service operation -->
    <name>getEmp</name>
    <!-- SQL code for a query -->
    <statement>select ename from emp
      where ename=:{myname VARCHAR}</statement>
  </operation>
  <operation>
    <!-- Name for the Web Service operation -->
    <name>updateEmp</name>
    <!-- SQL code for a DML statement -->
    <statement>update emp SET sal=sal+500
      where ename=:{myname VARCHAR}</statement>
  </operation>
</sql-statement>
```

Here we specified two SQL statements: one query and one DML statement. These will be turned into corresponding Web Service operations `getEmp` and `updateEmp`, respectively. The bind variables `{myname VARCHAR}` in the SQL statements will be turned into corresponding string parameters of the Web Service operations. Note that the query is returned as a result set, which can be materialized as an array of structured row values or in various generic XML formats, as discussed in the section on mapping of the SQL REF CURSOR type.

You will likely wonder about the semantics of SQL DML statements invoked as a Web Service. Since Web Services in general—and Database Web Services in particular—are considered to be stateless, the generated JPublisher code must automatically either commit a successful DML operation or roll it back in the case of failure. In addition JPublisher also generates corresponding batch-DML statements that accept an array of inputs for each of the parameters. You would utilize these entry points for achieving optimal performance if you require repetitive execution of a DML operation. Batches will automatically either be committed or rolled back in their entirety as well. Both, single as well as multiple-update operations return information on success or failure and—if the former—on the number of rows affected.

**PUBLISHING NATIVE SERVER-SIDE JAVA INVOCATIONS AS WEB SERVICES**

Usually you would base your Web Service on a Java class that is running in the Application Server middle tier. What can you do if you want to base your service on Java code that is running in the Oracle Database JavaVM? While it is possible to expose your code through PL/SQL call specs and then publish that as a PL/SQL Web Service, it is more convenient to expose the Java class directly as a Web Service. In this case you would have to specify a tag similar to the following to describe the database resource in your Web Services Assembler configuration file.

```xml
<db-java>
  <!-- Server-side Java class to be exposed -->
  <name>foo.bar.Baz</name>
  <!-- List of methods to be exposed (optional) -->
  <method>method1</method>
  <method>method2</method>
</db-java>
```

JPublisher, and thus Web Services Assembler, can only publish Java methods that meet the following criteria. They must be public static methods. Any Java type that is used in a parameter, a return, or an exception must be serializable. Additionally, all such types must exist and be defined in the same way in both, client and the server Java environments. Taking advantage of native Java invocation simplifies use of Java array and Java Bean types in method signatures – there is no need to provide
Our discussion on Web Services call-ins focused primarily on capabilities in Oracle Application Server 10g, which provides a JAX-RPC based Web Services implementation. How can you access Database Web Services call-ins without this stack and without new Web Services Assembler functionality? Several workarounds are available.

- You can package your database resource, such as a Query, or DML code, or server-side Java code into PL/SQL functions or procedures. Subsequently, you can easily publish the package containing these functions and procedures as a Web Service. Also note that with Oracle Application Server 10g [9.0.4], JPublisher supports PL/SQL legacy types, easing most of the Java/JDBC restrictions for accessibility of PL/SQL-only types.

- In [1] we describe how you can leverage the 9i Web Services Assembler to publish Java classes that are based on database connections as Web Services. As noted above, the JPublisher tool can –in turn– be enlisted to generate the code for such classes. Fundamentally, you can view JPublisher as a code generation tool for creating Java code that utilizes a JDBC connection to create class instances. When viewed this way, JPublisher can be employed with different Web Services stacks. One just needs to answer in each case how database connections will be handled, and how Java types are mapped to corresponding XML types. Internally JPublisher employs code generation templates that give it flexibility to create Apache-flavored Java classes in 9.0.4 and JAX-RPC-flavored Java classes for 10g that can both be utilized directly by their respective Web Services stacks.

**Deploying and Testing Your Web Service**

You will use Enterprise Manager, or the DCM control tool (or in a stand-alone OC4J installation the admin.jar tool) to deploy the EAR file that is generated by Web Services assembler. When you visit the deployed Web Services endpoint in your HTML browser, you are directed to the Web Services test page. Here you find a list of available operations and can also test them in your browser. Additionally, by appending “?wsdl” you can retrieve the WSDL specification of your service, and the suffix “?proxy_jar” lets you obtain the static Java client code that may be incorporated into a Java application to invoke the Web Service.

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

Oracle JDeveloper automates the creation of Web Services based on PL/SQL packages. You can 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 will guide you through the steps for publishing your Web Service.

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

With Oracle Enterprise Manager you 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, utilize Single Sign-on, and centrally manage all aspects of security. Moreover, Oracle Enterprise Manager also permits you to publish your Web Services to the Oracle UDDI Registry.
FROM DATABASE WEB SERVICES TO DATABASE GRID SERVICES – A ROADMAP

The Database Web Services described here do provide flexibility in the form of bind variables. However, they do not permit one to specify arbitrary SQL statements. While it would be straightforward to add syntax for such a capability, the following should be considered.

- Traditional database interactions are *stateful*. Typically, a database session will be established and subsequent database operations are issued against that session. The database and database sessions can both be regarded as *services* or *resources*. However, this is exactly the starting point for Grid Services. The Grid standards specify how stateful services can be modeled based on an underlying Web Services architecture and how the properties of resources can be described.

- More importantly, the Database Access and Integration (DAIS) group within GGF is working on an XML-based generic specification for database access, comparable to the JDBC specification but based on language- and platform-independent XML Schema representations of values and operations. This effort represents a generalization of Database Web Service, the main difference being that the DAIS effort is rooted in Grid Services rather than Web Services.

As Web Services stacks evolve to provide required capabilities and support for Grid Services, we also expect Database Web Services to assume more and more the full functionality of Database Grid Services. As an intermediate stepping stone to this we view the support of Grid Access operations from within a Web Services runtime stack.

However, there is no reason to wait for the arrival of full-blown Grid implementations to leverage the capabilities of Database Web services. Already you have seen how you can access all kinds of database APIs such as SQL, XML, PL/SQL, and Java through interoperable Web Service call-ins. And you know that from the database itself you are able to invoke and inter-operate with external Web Services, such as those based on J2EE and .NET.

CONCLUSION

We have provided an introduction to the main concepts of Web Services and Grid Services. We then explored in detail how external Web Services can be invoked from Java, SQL, and PL/SQL code inside the Oracle Database. Next, we examined several database resources and how they can be published as Database Web. The heart of the Data Access capability for Grid Services is provided by Database Web Services. We expect Database Web Services to evolve into full Grid Services, providing efficient, platform independent Data Access in the Enterprise Grid.

We invite you to visit the Oracle Technology Network [15,16] for more information on Database Web Services and Grid Services.

REFERENCES


