Deploying Enterprise JavaBeans™ in the Oracle8*i*™ Server

An Oracle Technical White Paper
April 1999

ORACLE

INTRODUCTION

Two factors are driving fundamental changes in business applications. The first concerns how these applications are developed. The second concerns how these applications are deployed. On the one hand, the changes are being driven by the rapid growth of business on the Internet, which has created new business models, reduced time to market, and significantly changed the way corporations look at developing and deploying Enterprise applications. On the other hand, the Internet also provides the medium for accelerated adoption of distributed computing. It provides standard protocols, like HTTP and IIOP, built on TCP/IP using the World Wide Web browser as its interface, and it allows universal access to these applications. The Internet has created several new classes of applications, such as electronic commerce and self-service applications, that need to be integrated with traditional Enterprise applications.

Today's IT architectures and IT applications need to adapt to rapidly changing business conditions in a global economy. The traditional methods of developing and deploying applications cannot satisfy the needs of most IT departments.

What's wrong with the traditional way of developing applications? It is not very productive. First, traditional applications do not lend themselves to code reuse. Therefore, significant pieces of code have to be rewritten to adapt to new applications. Second, several studies have shown that deploying applications in client-server configurations has led to a tremendous increase in the cost of ownership of these applications. Upgrading and maintaining client-server applications are very expensive tasks. To tackle these challenges, more and more IT organizations are using *component-based development* for new applications, and the *network computing* model to deploy these applications. There are significant advantages to using these approaches.

This paper discusses why businesses are using component-based development, and how to develop and deploy Enterprise JavaBeans-based applications in an Oracle environment.

Enterprise JavaBeans (EJB) is an architecture for component-based distributed computing. Enterprise JavaBeans are components of distributed transaction-oriented Enterprise applications. This paper is a part of a series of Java™ technical papers from Oracle that describe the architecture and programming models of its Java Virtual Machine (VM).

The paper is divided into five sections:

1. A discussion of why component-based development is becoming the preferred way of developing new applications. It discusses the role of Java in accelerating the adoption of component-based development.

2. An overview of Oracle8*i*'s integrated Java VM, Oracle JServer, and how it supports two different component programming methodologies, CORBA and Enterprise JavaBeans, to deploy Java applications in a scalable and reliable manner.

3. A description of how to deploy an Enterprise JavaBeans in the Oracle8*i* server. It outlines the necessary steps to develop, package, and deploy an EJB in the Oracle database.

4. A discussion of the interoperability between Java and SQL, and how components can access persistent data from the Oracle database.

5. Finally, a discussion of the benefits of using Enterprise JavaBeans, particularly the benefits of deploying them in the Oracle8*i* server.

COMPONENT-BASED DEVELOPMENT

The manufacturing industry long ago learned the benefits of assembling products from pre-fabricated components, rather than building custom products from scratch. One of the reasons behind the success of the PC industry was the ability to build desktop machines cheaply from off-the-shelf components that could be plugged together with ease.

Making components of any technology helps in the availability of low-cost, high-quality products. It also helps reduce time-to-market, by building from pre-built and tested components. This concept also applies to software. Component-based software development offers several benefits that make application development more productive:

- *Better Design And Code Reuse* — It allows users to reuse business logic. Instead of attempting to reuse applications at the class-level, server-oriented business logic is much more effectively captured as coarse grained reusable components.

- *Easier To Maintain* — It improves applications, since they are structured as reusable modules with clean interfaces.

- *Easier To Deploy Across Tiers* — It simplifies deployment because components can be transparently distributed across networked servers in a multi-tier environment. This allows considerable deployment flexibility.

Despite several attempts by the software industry, the promise of assembling business applications from pre-fabricated components has remained unfulfilled, until recently. Object oriented programming languages promised much reuse, but failed to deliver, because they did not enable the interoperability of components at the binary level. However, there is renewed hope that application developers will be able to assemble applications from pre-fabricated components. Component technologies, such as JavaBeans, CORBA, and Microsoft's COM, provide the infrastructure that enables developers to assemble applications from components that different vendors have developed.

Over the last few years, we have witnessed the rapidly increasing popularity of client-side components. JavaBeans and COM components are already being widely used to assemble GUI-based applications. A large number of vendors supply tools for developers, to help them rapidly assemble client-side applications, using beans or ActiveX controls. However, there was still a lack of a robust, scalable server-side component model.

Server components encapsulate business logic that can be deployed on a variety of servers that provide services, like transactions, security, and the messaging needed to manage these components. Server components are easily adaptable to changing business rules, and can be deployed across a variety of servers for high availability and scalability. Unlike client-side components, server-side components allow developers to rapidly assemble Enterprise applications from pre-fabricated components. These components are typically transactional, and need to run in a server environment that can scale to meet the challenging needs of today's mission-critical applications.

Three models are rapidly becoming popular for server-side components:

- *Enterprise JavaBeans (EJB)* — A cross-platform, server-side component model for Java, Enterprise JavaBeans extends the JavaBeans architecture to provide a distributed component architecture. It enables developers to focus on developing business logic, and packaging them as components that can be transparently deployed in servers that support the EJB platform. Developers do not need to worry about low-level system programming using session management, remote invocation, transaction, security, and multithreading. EJBs also enable developers to develop applications on one platform without worrying about how and where the EJBs will be deployed.

- *CORBA* — CORBA is a standards-based distributed component model proposed by the Object Management Group (OMG). It supports a development environment for building, deploying, and managing distributed object applications that are interoperable across platforms. CORBA objects communicate using OMG's Internet Inter-ORB Protocol (IIOP), the standard for communication between and among distributed objects running on the Internet, intranets, and in Enterprise computing environments. CORBA components written in different languages, running on different platforms, can transparently communicate and interoperate.

- *Distributed Component Object Model (DCOM)* — A proprietary distributed object model proposed by Microsoft for the Windows platform. DCOM extends Microsoft's COM component model, developed for building components that interoperate on a desktop to work across a network.

All three of these component models rely on development and deployment of applications based on a multi-tier, distributed object architecture. In this paper we show you how to develop and deploy EJBs in the Oracle8*i* database server.

ORACLE JSERVER VIRTUAL MACHINE

Java's promise of "write once, run anywhere" has made it the language of choice for developing and deploying intranet/Internet applications. More than 400 applications have already been certified as 100 percent Pure Java™. Java is enabling network computing solutions that can execute on a simpler, lower cost, network-centric IT infrastructure. While Java is helping IT lower the cost of ownership of business applications, it is also a very popular choice among application developers, because of its productivity benefits. Apart from being a modern, object oriented, platform-independent, and safe, Java offers a component model that enables application developers to build applications from components. Several tools available in the market today allow building client-side Java applications by dragging and dropping JavaBeans. However, until now, little progress has been made in leveraging these strengths to develop server-side applications. To succeed as a language for developing server-side applications, the Java Virtual Machine (VM) needs to be scalable, secure, manageable, and highly available.

JAVA FOR ENTERPRISE APPLICATIONS — A SERVER LANGUAGE

Java's initial popularity came from its ability to add dynamic content, in the form of applets, to web pages. However, Java is rapidly evolving toward its primary use in the future, writing Enterprise and Internet applications that can be deployed in either client-server, or intranet/Internet configurations. Several factors account for Java's popularity as a server language.

Java is a safe language and is, therefore, ideally suited for database integration. The database is a safe environment that provides the foundation for several mission-critical applications. Therefore, Oracle does not allow application code that could compromise the integrity of the database (code written in C) to run inside the database. Java is a strongly typed language, has native support for arrays and

strings, and has built-in support for memory management. The garbage collection mechanism built into the Java language frees developers from allocating and de-allocating memory, thereby eliminating problems of memory corruption and memory leaks.

Apart from being a safe language, Java enables true application partitioning in a multi-tier environment. Since Java is platform-independent, application logic developed in Java can be deployed on any server that supports Java. Applications written in Java can be easily used on another server, without having to be rewritten. Enterprises can leverage these benefits to improve time-to-market, and to lower system development and administration costs.

Although Java is an ideal language for developing Enterprise and Internet applications, there are no Enterprise class Java servers available in the market today. Most server-side Java applications run with JavaSoft's Java Virtual Machine, or some variant of it. The Java VM from JavaSoft was developed as a client-side VM that focuses on a single-user environment. It was not designed to run Enterprise applications, and it does not meet the requirements of an Enterprise server. It uses built-in multithreading to provide limited scalability. It does not offer the performance, scalability, robustness, and high availability needed for deploying Enterprise applications. Client-side VMs do not scale for several reasons:

- *Single Object Memory* — All objects are created in one type of "object memory," and, thus, need to be scanned by the garbage collector, which slows down performance and results in poor scalability.

- *Multi-User Scaling Via Threads* — The JDK Java VM achieves multi-user scalability, by using lightweight threads. This model does not scale very well when the number of concurrent users becomes very large, to the order of 10,000+.

- *Dynamic Compilation* — Java is an interpreted language, so it is slow. Several vendors have tried to improve the performance of Java programs, by using clever optimizations and just-in-time (JIT) compilers that compile the bytecodes just before execution. Since JIT compilers compile a class at a time, as they are downloaded into a browser, they provide limited performance improvements. JIT compilers focus more on fast compilation techniques, and less on the performance of the compiled applications. Again, these optimizations do not work well in a server environment, where most of the code that needs to be optimized is known much before deployment time. In this case, coarse-grained optimizations, beyond the reach of JIT compilers, can be used to improve performance.

Implementing a scalable Java Virtual Machine is a challenge, because it has to deal with automatic storage management, multithreading, and dynamic loading. Oracle JServer Java VM is compliant with Sun's Java specification, to provide the best-in-class server platform on which to deploy Enterprise and Internet Java applications.

Oracle JServer VM was designed with the following features:

- *High Performance* — For transaction processing and decision support applications
- *Scalability* — To support very large numbers of users
- *High Availability* — To meet the 24 x 365 availability of Enterprise and Internet applications
- *Manageability* — To lower the total cost of ownership, and deliver high quality of service
- *Compliance With Standards* — To enable any standard Java program to be deployed in the database

With the Oracle8*i* release, Oracle has delivered Oracle JServer VM, a server side Java Virtual Machine that is 100 percent compatible with JavaSoft's specification. It also meets the requirements for deploying mission-critical server-side applications: it is a highly scalable, highly available, easily manageable, secure and high performance Java Virtual Machine. Using Oracle JServer, businesses can realize Java's true potential, by developing business logic in Java and deploying it on a scalable Java platform. The Oracle JServer VM runs in the database engine (Figure 1), giving developers the flexibility of creating their applications in a variety of programming models. While application developers can still use the conventional stored procedures to develop their application logic, developers familiar with component-based programming can write their applications as CORBA objects, or as Enterprise JavaBeans, and deploy them in Oracle's Java platform.

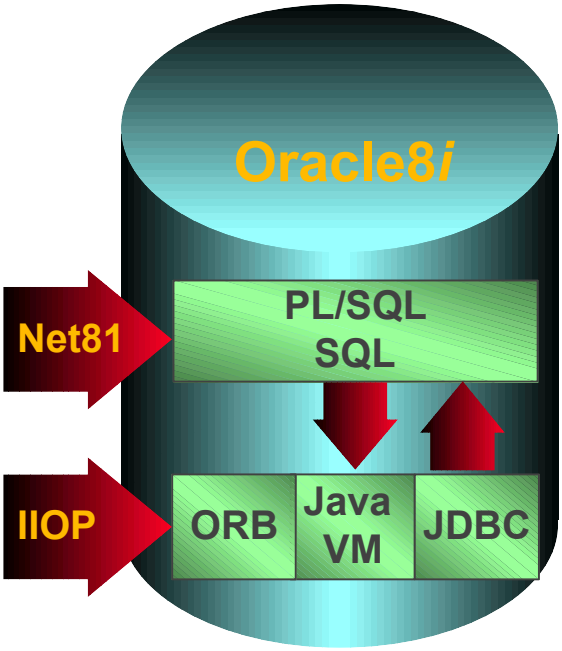

Figure 1. Oracle JServer Java VM

Java is also suited to completely new classes of software being written by independent software vendors to provide many new services on the Internet. Oracle's Enterprise Java platform will provide an ideal deployment vehicle for such applications, ensuring the widest availability of application solutions to Oracle's customers.

COMPONENT-BASED DEVELOPMENT USING ORACLE JSERVER

Developers can use two different component models to develop and deploy Java applications in Oracle8*i* Java Virtual Machine.

CORBA

First, developers familiar with CORBA programming can develop business logic as CORBA objects in Java, and deploy them in the Oracle8*i* server. Since the database has an embedded Java ORB and an IIOP interpreter, the Oracle8*i* server can be used as the infrastructure to deploy CORBA objects in a scalable manner. The database supports a variety of CORBA clients, including direct support for browsers, such as Netscape 4.0, and middle-tier ORBs.

Since the CORBA objects deployed in the server run in the same address space as SQL and PL/SQL™, they can seamlessly inter-operate and access SQL data very efficiently, without making any network round trips.

Enterprise JavaBeans

The other component model, predominantly used by Java programmers, is Enterprise JavaBeans. Oracle8*i* with Oracle JServer supports the standard EJB 1.0 specification, but provides a number of database specializations that ensure its implementation is fast, highly scalable, and secure. Here are some salient features of Enterprise JavaBeans:

- *Higher Level of Abstraction* — Enterprise JavaBeans offer a higher level of abstraction than CORBA, which does not require advanced systems programming skills.

- *No Foreign IDL* — Enterprise JavaBeans are simplest for Java developers, since they have a pure Java definition that avoids dependency upon systems, such as CORBA and DCOM. Java developers do not have to learn another interface definition language (IDL) to define the interfaces of the components they are developing.

- *Execute In a Container* — An EJB executes inside a container. Simply stated, a container provides an operating system process, or thread, in which to execute the component. Client components normally execute within some type of visual container, such as a form or a web page. Server components are non-visual and execute within a container supplied by an application execution environment, such as a transaction processing (TP) monitor, a web server, or a database system.

- *Declarative Programming Model* — Further, EJB has a simple declarative transaction model that allows users to specify transactional boundaries declaratively. They happen automatically when JavaBeans methods are executed.

- *Portable Across Servers* — In addition, EJBs are portable across a range of Java VMs. The EJB transaction server concept allows platform vendors to add value in the form of scalability, reliability, and atomic transactions.

To support component-based programming, Oracle has integrated two main components with the Oracle8*i* release:

CORBA 2.0 Compliant Java Object Request Broker (ORB)

The SQL-oriented listener/dispatcher architecture has been extended to support CORBA/IIOP. The ORB is used only for IIOP protocol interpretation and object activation, while the database's multithreaded server (MTS) kernel is leveraged to deliver scalability. The MTS architecture has previously demonstrated scalability into the tens of thousands of concurrent users. Thus, our purpose was to support open-systems Java APIs on an existing, highly scalable infrastructure. The Java ORB enables users to call into and out of the database server using CORBA's IIOP protocol. CORBA objects deployed in the Oracle8*i* database can be invoked using the industry standard IIOP protocol. The database server then behaves as a CORBA server. Similarly, IIOP callouts from the database allow it to serve as a standard CORBA client and invoke CORBA objects in other servers. Further, the CORBA facilities in the RDBMS are both Java-friendly with a number of automated tools, such as the automated "Caffeine" Java-to-IDL compiler.

Specialized Object Adapter

The database supports a specialized Object Adapter that has been implemented for persistent CORBA objects. It serves as a directory of all CORBA objects published in the RDBMS, locates and activates CORBA Objects, and provides access control. Details of how to implement CORBA objects in the Oracle8i server are discussed in a separate paper.

HOW TO DEVELOP AND DEPLOY ENTERPRISE JAVABEANS IN THE ORACLE SERVER

Now that we have described Oracle JServer and the component models supported in the Oracle environment, let's look at how to implement Enterprise JavaBeans in the Oracle environment. This section will give you complete step-by-step details on how to develop and deploy Enterprise JavaBeans in the Oracle8*i* database. Since Enterprise JavaBeans have pure Java definition, users do not need to use an IDL to define the object interfaces. All interfaces are defined using Java.

Before we dive into the EJB example, let's briefly overview some concepts regarding state, persistence, and the different types of Enterprise JavaBeans.

- *Persistent Data* — Data that resides in the persistent store. It is transactional and shared. Examples of persistent stores are a relational database and an object oriented database. Examples of persistent data are a row in a table of a relational database or a page of objects in an object oriented database.

  The transactional sharing of persistent data is handled by the persistent store, and the transactional services of the EJB runtime are used to ensure the ACID properties of the persistent data.

- *Transient Data* — Data that resides only in the Enterprise bean itself. Transient data is not reflected in the persistent store and is private to the Enterprise bean instance.

- *The Persistent State Cache* — The cache that the Enterprise bean uses to manipulate the data in the persistent store. Thus, the persistent cache represents persistent data, but is not the persistent data. The developer uses hooks that demarcate the beginning and end of a transaction to provide a coherent view of the persistent data.

- *The Conversation State* — A composite of both the cached persistent data and the transient data.

The Enterprise JavaBeans specification describes three types of beans (*stateless session beans, stateful session beans*, and *entity beans*), primarily related to how the beans persist in their state. The EJB container manages the state of the bean that is referring to the data contained within the bean.

Session Beans

A session bean is created by a client, and in most cases exists only for the duration of a single client-server session. A session bean performs operations on behalf of the client, such as accessing a database or performing calculations. Session beans, also known as transient beans, can be transactional, but they cannot be recovered after a system crash. There are two kinds of session beans:

- *A Stateless Session Bean* — A bean that maintains no conversational state across methods and transactions. The EJB server transparently reuses instances of the bean to service different clients.

- *A Stateful Session Bean* — A bean that maintains conversational state across methods and transactions. As a result, the EJB server binds the Enterprise Bean instances to clients directly. Session beans, by definition, are not persistent, although they may contain information that needs to be kept. Session beans can implement persistence operations directly in the methods in the Enterprise bean. Session beans often maintain a cache of database information that must be synchronized with the database when transactions are started, committed, or aborted. With Oracle8*i*, such state synchronization occurs using JDBC or SQLJ. SQLJ is a standard way of embedding static SQL statements in Java programs.

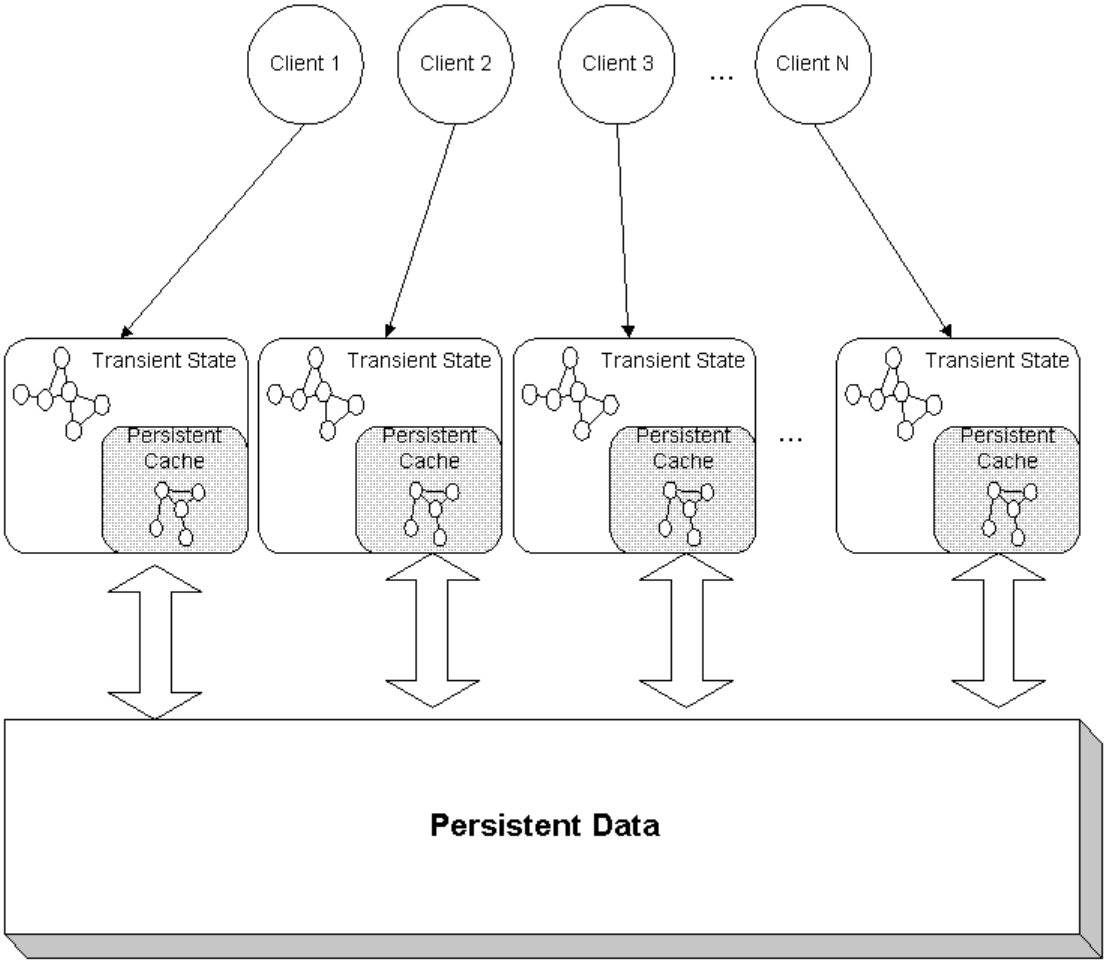

Figure 2. How Clients Access Session Beans

Entity Beans

Also known as persistent beans, entity beans have an identity that, in general, survives the crash and restart of the container in which they are created. Transparent to the client, the container provides security, concurrency, transactions, persistence, and other services for the EJB objects that live in the container. Multiple clients can access an entity object concurrently. The container in which the entity bean is installed synchronizes access to the entity state.

Entity Beans are an optional part of the Enterprise JavaBeans specification, and this part of the specification has not been completely standardized. As a result, Oracle8*i* does not support them. It only supports Session Beans. Oracle plans to support entity beans in a future release.

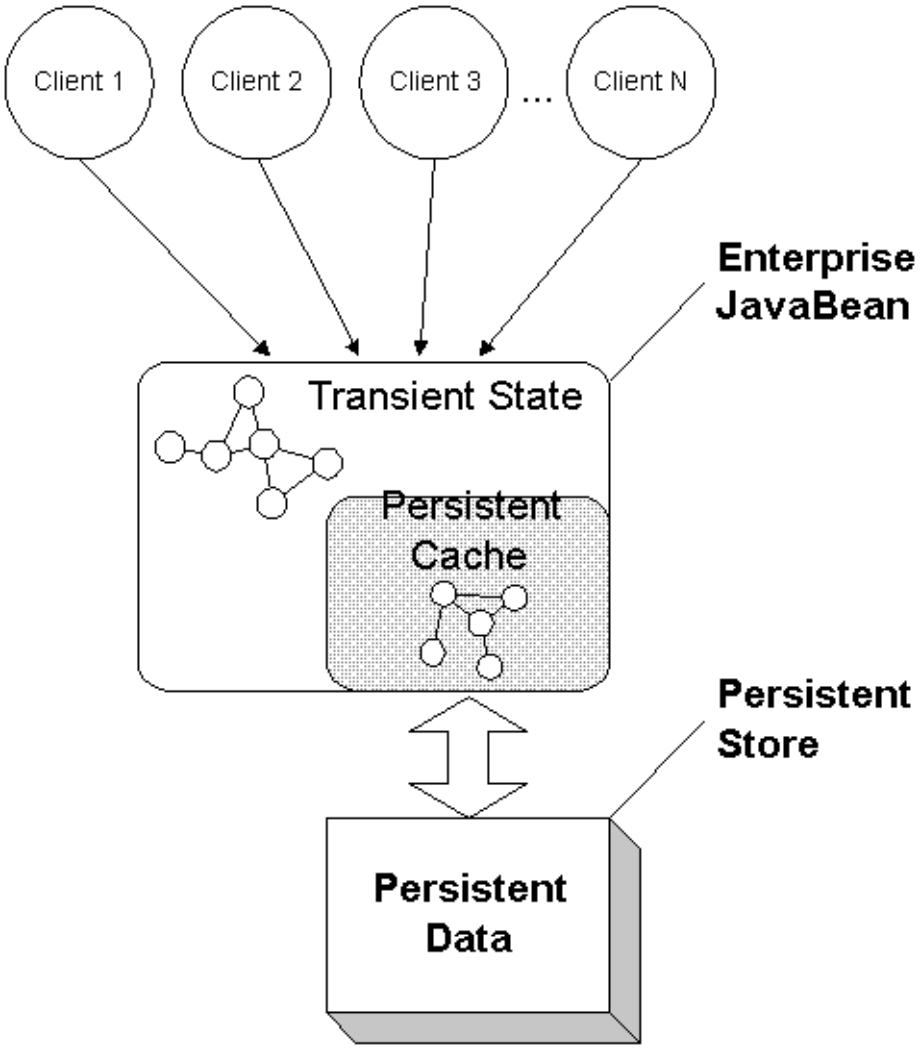

Figure 3. How Clients Access Entity Beans

WRITE YOUR JAVA CODE

Let's now look at the steps involved in developing Enterprise JavaBeans. A developer needs to define four items for client applications to be able to invoke methods of the bean that contain the business logic. We present this section using the familiar bank account example. The account bean we are using in this example serves two main functions. It can tell you the balance of the account and it can debit the account.

1. Define Your Remote Interface

Since Enterprise JavaBeans can be invoked remotely, the first thing that a developer needs to do is define the remote interface of the EJB. The remote interface lists all the methods or public interfaces of the bean that clients can call. While the bean remote interface is defined as a standard Java RMI (Remote Method Invocation) interface in the EJB specification, the actual communication is independent of RMI. Oracle8*i* uses IIOP as the wire protocol for communication. The remote interface definition is similar to the IDL definition of a CORBA object. Following is an example of how you can define the remote interface for the account bean:

```
// Bean Remote Interface extends javax.ejb.EJBObject
// All methods throw java.rmi.RemoteException
public interface Account extends javax.ejb.EJBObject {
  float getBalance () throws java.rmi.RemoteException;
  void debit (float amount) throws java.rmi.RemoteException;
}
```

2. Define Your Home Interface

Because beans can be invoked in a remote server, just using Java 'new' is not sufficient enough to create objects in a remote server. Beans have to provide a mechanism for clients to be able to create an instance of the bean. Conventionally, these interfaces are called object or bean factories. The home interface serves as the factory interface of the EJB. A home interface defines zero or more *create(...)* methods, one for each way to create an EJB object. Clients invoke this factory interface to create instances of the bean. Developers only need to define the factory interfaces, and not implement any methods of the factory interface. At deployment time, the server-side implementation classes for the bean home interface, which are EJB server-specific, are generated automatically. With Oracle8*i*, they are generated when the EJB is loaded into the database through an automated facility Oracle provides.

```
// Bean Home Interface extends the javax.ejb.EJBHome interface
public interface AccountHome extends javax.ejb.EJBHome {
  Account create (String account) throws
java.rmi.RemoteException,
javax.ejb.CreateException;
}
```

3. Define the Bean Itself

The bean itself is a standard Java application that implements the business logic written by the bean developer. The user does not need to describe the bean's transactional and security semantics in the application code. They can be described declaratively, using the bean's deployment descriptor. Since Oracle8*i* supports session beans, users need to explicitly persist the state of their bean. Since the Oracle JServer EJB server is tightly integrated with the Oracle RDBMS, users manage the bean's persistent state via JDBC or SQLJ.

```
// The Bean Itself - This is the place where you implement all
your business logic.
public class AccountBean implements javax.ejb.SessionBean {
  // Bean Interface methods
  public float getBalance () {
// ADD YOUR IMPLEMENTATION HERE
…
}
  public void debit (float amount) {
// ADD YOUR IMPLEMENTATION HERE
…
}
  // Home Interface methods.  These methods should have a 1:1
correspondence with the methods
  //  defined  in  the  home  interface.    These  methods  are
implemented by the application developer.
  public void ejbCreate (String account) {…}
}
```

4. Define the Bean Deployment Descriptor

Once you have defined the above interfaces and classes, that's all the code you need to write for your beans. The bean deployment descriptor ties the various components of the bean together. The deployment descriptor is a Java serialized object that extends the base deployment descriptor class. It allows the user to specify the bean's transactional and security attributes declaratively, simplifying the process of building transactional applications with the database. The bean attributes can be specified at the bean level, or the user can use method descriptors to specify finer grained transaction and security at the method level of the bean. Oracle offers special syntax to specify the deployment descriptor for beans. The EJB specification requires that the deployment descriptor be stored as a serialized object. It is tedious to create a serialized instance of a deployment descriptor, so our tools accept the deployment descriptor as text, and create the serialized object automatically. The syntax of the textual form has been designed to look like a Java class, and begins with a SessionBean statement. The statement is followed by the full qualified class name of the Enterprise JavaBean.

The descriptor is a list of bean attributes, which have the general form:

```
<attribute-name> = <attribute-value>;
```

There are three types of bean attributes:

- *Attributes of the Bean Itself* – For example, its transaction and security attributes, JNDI name, and the name of its remote and home interfaces. Some attributes are required (for a detailed list of required attributes, see the Oracle documentation).

- *Attributes of the Bean Methods* – The transaction and security related attributes can also be specified for individual methods of the bean. In the textual form, you identify the method signature, using standard Java syntax, but you replace the method body with a set of method attributes. The method qualifiers, such as public or static, the method return type, and the method argument name are optional and ignored.

- *Environment Properties For the Bean* – You can also specify the Bean environment properties, with an entry named EnvironmentProperties inside the bean descriptor. The parser builds Property Object from the list, and makes it available as the EnvironmentProperties attribute of the bean descriptor.

Here is an example of how you can create the deployment descriptor:

```
// Bean Deployment Descriptor for Account Bean
SessionBean AccountBean {
  BeanHomeName = "bn=myAccount";              //        Required
Attribute
  RemoteInterfaceClassName = Account;              //  Required
Attribute
  HomeInterfaceClassName = AccountHome;       //        Required
Attribute
  AllowedIdentities = {PUBLIC};
  SessionTimeout = 3000;
  StateManagementType = STATELESS_SESSION;
  TransactionAttribute = TX_MANDATORY;
  IsolationLevel = TRANSACTION_READ_UNCOMMITTED;
       public void getBalance ()
        {
             RunAsIdentity = WENDY;
        }
      public void debit (float amount)
        {
            AllowedIdentities = {TELLER};
            TransactionAttribute = TX_REQUIRES_NEW;
        }
EnvironmentProperties
  {
     MinInitialBalance = 100.00;
```

```
    }
                        }
```

COMPILE AND PACKAGE YOUR JAVA CODE

Once you have written the classes, interfaces, and deployment descriptor cited above, you need to perform the following steps:

1. Compile Your Code

You must compile the EJB itself, its remote interface, and its home interface to Java classes.

```
// Compile the Bean, RemoteInterface, HomeInterface to Java
classes
% javac   AccountBean.java   Account.java   AccountHome.java …
```

2. Package the Class Files

Next, you need to package the EJB, so that it is ready to be deployed into Oracle8*i*. Packaging the EJB essentially requires you to archive the compiled class files for the EJB itself, the remote interface, and the home interface in an EJB package — a standard Java jar file.

```
// Package the class files in an EJB package
% jar   cf   Account.jar    *.class
```

3. Add the Deployment Descriptor

Finally, you must add the EJB deployment descriptor to the bean, specifying its transaction and security attributes declaratively. To do so, you must use the command-line "createejb" tool to add the descriptor to the jar.

```
//  Add  the  EJB  Deployment  Descriptor  specified  in  the
Account.dsc file
// Use the createejb tool supplied by Oracle
% createejb   Account.jar  -desc  Account.dsc
```

DEPLOY THE EJB INTO THE ORACLE DATABASE

After packaging, you are ready to deploy Enterprise JavaBeans into the Oracle8*i* server. To deploy EJBs on Oracle8*i,* you need to use the "deployejb" tool that Oracle provides. This tool automates the process of deploying EJBs on Oracle8*i*, making it a single-step deployment. The tool automates three related actions:

1. Generating Server-Side Runtime

First, the tool automatically generates EJB server-side runtime. The tool automatically parses the EJB deployment descriptor, and generates the information that the Oracle JServer EJB server requires for declarative security and transactions. It then uploads the server-side runtime into Oracle8*i*. Since the EJB itself has been packaged as a standard Java .jar file, this process is automated by using the LOADJAVA utility that Oracle provides.

2. Generating Client Side Classes

Second, the tool automatically generates the client-side classes required to access the EJB server. It implements server-independent interfaces, and outputs them in a new jar file (Accountclient.jar).

3. Performing Additional Set-Up

Third, it executes the additional steps required to deploy the EJBs in the database automatically. Setup requires the following procedures:

- Populate the JNDI-accessible name service described in EJB's deployment descriptor
- Enforce security for the RunAs properties of the Bean
- Generate communication support code

Here's an example of how you can deploy the EJB:

```
// Deploying EJBs using deployejb tool
// user name is the schema in which the EJB will be deployed
and service name is the service context
// in which the Bean will be deployed.
% deployejb   -u  <user name> -p <password>  -s <servicename> -
gen   Accountclient.jar   Account.jar
```

WRITE YOUR EJB CLIENT

Once you have successfully completed the steps mentioned above, your EJB is deployed and you can activate it from a client. Writing a client for this EJB requires three steps.

1. Authenticate the Client

The EJB is deployed in the database, which is a secure environment. Therefore, before you can activate any EJBs you must first authenticate the client with the Oracle8*i* server. There are several ways to do this:

- *Explicit Authentication* — The client can authenticate to the database by first looking up the 'Login' bean. The 'Login' bean is the only bean that a client can access before activating any other beans. Then the client can call its authenticated method and provide a username, password, and role. Once the 'Login' bean has successfully logged the client into the Oracle database, the client can use other beans to which it has access rights. Any Java clients can use this mechanism.

- *Implicit Authentication* — To access an Enterprise JavaBean, the client can also initialize a 'Credential' object with its username, password, and role. These are sent to the server with the first IIOP message. This approach is similar to passing transaction contexts around. The server, upon receiving the credential, checks the username/password with the database, and does not allow any further access until a valid credential has been verified.

- *SSL-Based Authentication* — In addition to password-based authentication, Oracle's Enterprise JavaBean container optionally provides secure socket layer (SSL) support. This feature allows stronger authentication than the password-based authentication. It also provides privacy and data integrity, which the password-based authentication cannot do. Currently, Oracle only supports encryption and server-side authentication with SSL.

2. Locate the Bean's Home

Once the user has been authenticated with the database, the next step is to locate the bean's home and access it via standard bean interfaces. Clients locate the bean home from the standard directory or name service via JNDI.

3. Create Instances of the Bean and Activate Methods

Clients can now create instances of the bean by invoking the bean's home interface. The home interface serves as a bean factory, generating instances upon requests from the client. Once bean instances are created, users can execute methods of the instance by invoking the Bean's methods, published in the remote interface. Clients get the bean's remote interface from the .jar file generated at EJB deployment time. Clients can control transactions in two ways, either depend on the

declarative transactions specified as part of the EJB's deployment, or use Java Transaction Service (JTS) to explicitly drive transactions from the bean client.

Here is an example of a client using the Credential object to authenticate with the database:

```
// Implement EJB Client accessing the server bean
// PLEASE NOTE:  IN ORDER TO KEEP THE CODE BRIEF WE ARE NOT
// CATCHING ANY EXCEPTIONS.  YOU WILL HAVE TO ADD THEM IN YOUR
CODE
public class AccountClient {
  // Authenticate using Credential Object
public float debit_account(float amount, String aname) {
String uname = "scott";
String password = "tiger";
Credential.authenticate("uname","password");
// Create a new Initial context for use with JNDI
      InitialContext ctx = new InitialContext (…);
 // Create a Bean instance using the Home Interface
      AccountHome   ahome   =   (AccountHome)   ctx.lookup   ("…
bn=myAccount");
//  Now  that  you  have  the  Account  factory,  you  can  create
Account objects using the create method
Account new_account = ahome.create(aname);
// Invoke the Bean's interfaces
new_account.debit(amount);
float balance = new_account.getBalance ();
return (balance);
           }
           }
```

ACCESSING PERSISTENT DATA FROM ENTERPRISE JAVABEANS

So far, we have discussed how to develop and deploy Enterprise JavaBeans in the Oracle8*i* database. With the help of some simple examples, we demonstrated the steps needed to create, package, and deploy EJBs in the database. However, the examples we used were simplistic, and did not involve access to any persistent state in the Oracle database. In this section, we look at how EJBs access persistent data.

Enterprise JavaBeans running in the database can use JDBC or SQLJ to access persistent data. The Oracle Java Virtual Machine has a special version of a JDBC driver that runs in the database. This embedded JDBC driver complies with the JDBC 1.22 specification, and supports the same APIs that the client-side JDBC drivers support. Since this JDBC driver runs in the same address and process space as the database, it is optimized to directly access SQL without making any network round trips.

BENEFITS OF ENTERPRISE JAVABEANS

Enterprise JavaBeans offer several benefits over CORBA or DCOM programming and conventional procedural programming. First, let's look at the benefits offered by Enterprise JavaBeans. We will then consider benefits of using Oracle8*i* for deploying EJB applications.

EASE OF PROGRAMMING

For Java developers, EJBs are significantly easier to program than other component systems, such as CORBA or DCOM, for the following reasons:

- *No New IDL* — Since Enterprise JavaBeans have pure Java definition, users do not need to know foreign IDL. EJB supports the notion of Objects by value for any valid RMI type, allowing arguments to be passed by value, rather than as references.

- *Declarative Transaction Rules* — The transaction rules for an EJB can be defined at application assembly or deployment time. The transaction semantics are defined declaratively, rather than programmatically. The EJB server automatically manages the start, commit, and rollback of transactions on behalf of the EJB, according to a transaction attribute specified in the deployment descriptor associated with the EJB. The EJB specification offers a variety of choices of transaction policies: BEAN_MANAGED, NOT_SUPPORTED, SUPPORTS, REQUIRES, REQUIRES_NEW, and MANDATORY. This variety allows an EJB to adopt different types of behavior.

- *Declarative Security* — Similarly, the security policies on the EJB can be specified declaratively.

- *No System Level Programming* — Finally, the EJB developer does not need to deal with low level system programming issues, such as thread-aware programming. Scalability requirements are automatically addressed by the EJB server implementation.

EASE OF DEPLOYMENT

EJBs are very easy to deploy in a multi-tier, distributed environment:

- *EJB Clients Are Independent of Server Implementations* — The deployment process automatically generates EJB client classes and stubs. Further, clients implement server independent interfaces. As a result, they disregard the specific EJB server implementation chosen. Other elements, such as a declarative mechanism to specify security, allows the security attributes to be modified when the EJB is deployed.

- *Highly Customizable* — Similar to JavaBeans, the EJB component model supports customization without requiring access to source code. Further, the ability to specify transactions and security behavior declaratively allows them to be customized at deployment time.

SCALABILITY WITH LIMITED DEVELOPER EFFORT

EJBs allow server applications to scale with limited developer effort, since the EJB definition was specifically restricted to allow server scalability. The Enterprise JavaBeans environment automates the use of complex infrastructure services, such as thread management. Component developers and application builders do not need to implement complex service functions within the application programming logic.

EJBS ARE PORTABLE

Server logic defined as EJBs is portable to different EJB servers. EJBs were designed for portability across Java VMs. and EJB servers that comply with the standard EJB specification in three ways.

- *Standard EJB Package Format* —— All EJB servers accept EJBs in a standard format, called the ejb-package format. Oracle8*i* accepts packages in this format. They are generated by the 'deployejb' tool that automates deployment of EJBs on the Oracle RDBMS, and generates the server-specific code during loading.

- *Independent of Deployment Container* — The EJB model defines the relationship between an EJB component and an EJB container system. EJBs do not require the use of any specific container system. Any application execution system can be adapted to support EJBs, by adding support for the services defined in the specification. The services define a contract between an Enterprise bean and the container, effectively implementing a portability layer. The EJB itself and EJB client program are written to be independent of the EJB server on which they are deployed, and can,

therefore, be moved easily from one EJB server to another in a distributed environment.

- *Declarative Programming* — Since EJBs specify transactions declaratively, no explicit transaction code is required within the application itself. As a result, EJBs are portable across different transaction managers.

OPEN STANDARD SERVER COMPONENT MODEL

Finally, EJBs have tremendous momentum and widespread industry support as an open, distributed, server-oriented component model for Java. Most major vendors, including Oracle, IBM, Sybase, Netscape, and BEA Systems, have participated in the definition of the Enterprise JavaBeans specification. These vendors have indicated plans to implement support for Enterprise JavaBeans in their products. Further, EJB is an open, cross-platform component model, unlike a proprietary component model, such as Microsoft's DCOM, which is specified only for the Windows platform.

Now that we have examined the various benefits of EJBs, let's look at the specific benefits that Oracle8*i* offers EJB developers. Oracle8*i* makes EJBs:

SCALABLE

First, EJB components are standard Java applications that can exploit the Oracle JServer's shared memory optimizations and Oracle8*i*'s multi-threaded server infrastructure. This partnership achieves scalability to 10000+ concurrent clients. Second, immutable portions of the EJB, including its metadata and initial prototype, are placed in shared memory, only once, and shared across all EJB instances, no matter how many instances are running concurrently. Both these features, coupled with several other optimizations, lead to unprecedented scalability for Oracle JServer EJB server.

HIGH PERFORMANCE

Several features in Oracle's EJB server deliver high performance. Since EJBs are standard Java applications, they can make Java execution fast by leveraging the variety of performance optimizations that Oracle JServer provides.

- *Efficient Access To SQL* — Since EJBs use the embedded JDBC driver to access persistent state in the RDBMS, SQL data access is very efficient. Oracle8*i* provides an ideal EJB server platform for data intensive EJBs.

- *Oracle JServer Accelerator (Native Compilation)* — Since EJBs are standard Java programs, they can use the Java through C translator, and other Java VM optimizations to improve performance significantly.

EASY-TO-USE

EJB transactions map to traditional RDBMS transactions, providing a familiar and high-performance transaction coordination mechanism for users. Using EJB as a programming paradigm, users can now specify RDBMS transactions declaratively, rather than programmatically. EJB security maps to traditional RDBMS users and roles. Oracle, therefore, provides a highly secure EJB server.

INTEGRATED WITH TOOLS

The automated EJB deployment tools make it very easy to package and deploy EJB applications on Oracle8*i*. Further, these tools are being integrated with Oracle's JDeveloper™ IDE, to facilitate and simplify the application development process. Oracle also supplies a set of EJB server administration tools that allow users to perform various simulations and administration functions on server-deployed EJBs.

SECURE

A variety of security mechanisms can be used with Enterprise JavaBeans deployed in the Oracle8*i* database. Application developers have the flexibility of using the traditional user name/password-based authentication, SQL level security via traditional RDBMS privilege mechanisms, and network level encryption via SSL.

SUMMARY

This paper has given you a quick overview of how to develop Enterprise applications, such as Enterprise JavaBeans, and deploy them in the Oracle8*i* database. Oracle JServer, the Java engine integrated with the Oracle8*i* database provides an open server platform for deploying Java applications that is scalable, robust, secure, highly available, and easily manageable.

Enterprise JavaBeans provide a robust server-side component model that allows developers to build applications by assembling components from different vendors. This enables developers to quickly adapt applications to rapidly changing business needs. The declarative programming model of Enterprise JavaBeans enables developers to focus on business logic, not system programming issues.

In brief, there are four steps needed to use Enterprise JavaBeans in the Oracle database:

1. Define Your:
   - Remote Interface
   - Home Interface
   - Bean
   - Deployment Descriptor
2. Compile and Package Your Bean
3. Deploy the Bean in the Oracle Server
4. Write Your Client Application That Invokes Methods of the Bean

Since Java programs run in the same address space as the database server, and can leverage the embedded JDBC drivers, they need not make network round trips to access SQL data, and are, therefore, optimized to deliver high performance. Java is tightly integrated with the database, and can seamlessly inter-operate with SQL and PL/SQL.

Oracle's implementation of Enterprise JavaBeans offers a variety of benefits to developers, including high scalability, performance, availability, application throughput, and seamless interoperability among Java, SQL, and PL/SQL.

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
+1.650.506.7000
Fax +1.650.506.7200
http://www.oracle.com/