Developer's Guide to Understanding Enterprise JavaBeans™

Chris Crenshaw
Nova Laboratories
www.nova-labs.com

For more information about Nova Laboratories or the Developer Kitchen Series, or to add your name to our monthly mailing list, visit our website at http://www.nova-labs.com

Nova Laboratories also offers in-house and on-site training and consulting services.

Nova Laboratories
187 Monmouth Park Highway
West Long Branch, New Jersey 07764

732-263-9000
732-263-0189 (fax)

This material is copyrighted by Nova Laboratories © 1998. This material shall not be reproduced or distributed in any form without the express written consent of Nova Laboratories.

All rights reserved.

The Developer Kitchen is a registered servicemark of Nova Laboratories. Java and all Java-based trademarks and logos trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All other products referenced herein are trademarks of their respective holders. © 1998 Nova Laboratories.

1 - Preface

Purpose of this Document

The Enterprise JavaBeans™ 1.0 Specification is the formal specification for Enterprise JavaBeans™. It formally defines a set of standards for a Java™ server component specification. However, it tends to be targeted more toward the vendors who need a blueprint describing the mininal requirements for creating an EJB server than for developers who are looking to learn about EJB.

This document is not intended to replace the Specification. The Specification is the official blueprint for Enterprise JavaBeans™. Instead, this document will hopefully serve as a useful primer for developers who want a technical introduction to EJB. It attempts to summarize areas of the Specification that are of particular interest to developers who are trying to learn EJB.

Nor is this document a tutorial on EJB. Such a document could be several hundred pages in itself. Instead, its goal is to clearly lay out a detailed description of the architecture of EJB, and how the various components fit together.

Overall, the purpose of this document is to provide a solid technical introduction to EJB. It should benefit both developers looking to start writing with EJB as well as developers who need to perform a detailed analysis of EJB to determine whether an EJB solution is appropriate for their application.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

How to read this document

This document consists of three sections, plus this Preface. Additional sections will be added to this document over time that address some of the more specific advanced features of Enterprise JavaBeans™.

Preface - This section. Defines some basic terms that will be used throughout this document. Lists some available documents and products that may be useful to the reader.

Introducing Enterprise JavaBeans™ - This section more fully describes the purpose of this document. The high-level architecture of Enterprise JavaBeans™ is illustrated, along with simplified definitions of the major components. Also in this section are explanations of how Enterprise JavaBeans™ relates to other technologies, such as Java™ RMI, JavaBeans™, and CORBA® . Some of the benefits of Enterprise JavaBeans™ are listed here, along with definitions of the various roles that a developer or vendor may fill in creating an Enterprise JavaBeans™ solution.

Understanding EJB Components - In this section the reader will get very detailed definitions of the components of Enterprise JavaBeans™. The focus of this section is to explain how these components operate in a runtime environment. An attempt is made to describe the minimal behavioral requirements of the components as defined in the Enterprise JavaBeans™ Specification, along with some practical scenarios describing how vendors will likely implement these components.

Transactions - Transaction control is a powerful feature of Enterprise JavaBeans™, and can be a very complicated subject. This section will explain how transactions are used and controlled within Enterprise JavaBeans™. Also in this section is coverage of the CORBA® Object Transaction Service, or OTS. The transaction model in Enterprise JavaBeans™ is essentially an implementation of OTS, and understanding the interfaces and semantics of the components in OTS will allow the reader to better understand transactions in Enterprise JavaBeans™.

Defining some basic terms

EJB - a common abbreviation of Enterprise JavaBeans™.

developer - person who writes implementations of EJB classes.

vendor - creator of an EJB product that can be used to develop, install, or deploy an EJB class written by a developer. This term is generally used in this document to refer to an organization that has created a server product capable of running EJB classes written by an EJB developer.

EJB class - a Java™ class written and compiled by a developer that conform to the interfaces and requirements described in the Enterprise JavaBeans™ Specification.

EJB instance - an instantiation of an EJB class.

Specification - the Enterprise JavaBeans™ Specification

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

Additional terms will be defined at the appropriate times throughout this document.

Sources and references

Documentation

Nova Laboratories website containing this document, descriptions and availability of EJB lecture and workshop training, and source code examples:

`http://www.Nova-Labs.com`

Main page for Enterprise JavaBeans™:

`http://java.sun.com/products/ejb`

Enterprise JavaBeans™ Specification and other documentation:

`http://java.sun.com/products/ejb/docs.html`

JNDI - Java Naming and Directory Interface™:

`http://java.sun.com/products/jndi`

JTS - Java™ Transaction Service:

`http://java.sun.com/products/jts`

OTS - CORBAservices® Object Transaction Service:

`http://www.omg.org/docs/formal/97-12-17.pdf`

Trademarks

Enterprise JavaBeans™ , Java™, JavaBeans™, JDBC™, and Java™ Naming and Directory Interface™ are trademarks or registered trademarks of Sun Microsystems, Inc.

CORBA®, CORBAservices®, and IIOP™ are trademarks or registered trademarks of Object Management Group, Inc.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

This page intentionally left blank

 Copyright © 1999 Nova Laboratories - Reproduction Prohibited

2 - Introducing Enterprise JavaBeans™

Introducing Enterprise JavaBeans™

Services Framework

Enterprise JavaBeans™ is not a product. It is a specification for a Java™ server-side services framework from which vendors can create EJB server implementations. The benefit to application developers is that they can focus on writing the business logic necessary to support their application without having to worry about implementing the surrounding framework.

The Specification details certain minimal yet crucial services such as transactions, security, and naming. The vendors must follow these specifications to ensure that an enterprise bean can always rely on certain required services. The Specification does not state how vendors should go about implementing these services. While this can make it harder to learn EJB simply by reading the Specification, it does allows vendors the freedom to provide enhancements without sacrificing portability regarding core services.

JavaBeans™ vs Enterprise JavaBeans™

JavaBeans™ is the component model for Java™. Within the JavaBeans™ Specification there are features defined such as events and properties. Enterprise JavaBeans™ also describes, among other things, a Java™ component model, but the component model for Enterprise JavaBeans™ is not the same as JavaBeans™.

With JavaBeans™ the emphasis is on allowing developers to visually manipulate components in a builder tool. To that end, the JavaBeans™ Specification describes in detail the APIs and semantics for inter-component

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

event registration and delivery, recognition and utilization of properties, customization, and persistence.

The emphasis for Enterprise JavaBeans™ is to detail a model for a services framework into which Java™ components can be portably deployed. Consequently, there is no mention of events, since enterprise beans do not typically send or receive events. Nor is there mention of properties. Customization does exist, but is not performed at development time using properties, but at runtime (deployment time, actually) using a ***deployment descriptor***.

Do not look for similarities between JavaBeans™ and Enterprise JavaBeans™. They are both specifications for component models, but one addresses issues of application assembly in a builder tool, while the other details a services framework into which components can be deployed.

Don't make the mistake of thinking that JavaBeans™ is for client-side development and Enterprise JavaBeans™ is for server-side development. JavaBeans™ can be an appropriate component model for building non-graphical server-side Java™ applications. The difference is that when you use JavaBeans™ to create a server application, you have to build the entire server framework. With Enterprise JavaBeans™ the framework is provided for you; you simply conform to its APIs. For complex server-side applications it is easier to plug in than reinvent.

Enterprise JavaBeans™ Architecture

The EJB server is the high-level process or application that manages EJB *containers*, along with providing access to system services. The EJB server may also provide vendor-specific features, such as optimized database access interfaces, availability of additional services such as CORBAservices®, support for SSL 3.0, and so forth.

An EJB server is required to provide availability of a JNDI-accessible naming service and a transaction service.

Some examples of EJB servers may be:

- database servers
- application servers
- middleware servers

The EJB container is an abstraction that manages one or more EJB classes and/or instances. It makes required services available to the EJB classes through an interface defined in the Specification. The container vendor may also provide interfaces to additional services implemented either in the container or in the server.

There is currently no specification for the interface between the EJB server and the container. Therefore the container is currently provided by the EJB server. Once an interface is standardized, it is likely that vendors will create containers that can run in any conformant EJB server.

The *home interface* lists the available factory methods for locating, creating, and removing instances of EJB classes. The *home object* is the implementation of the home interface. The developer of the EJB class must also define the home

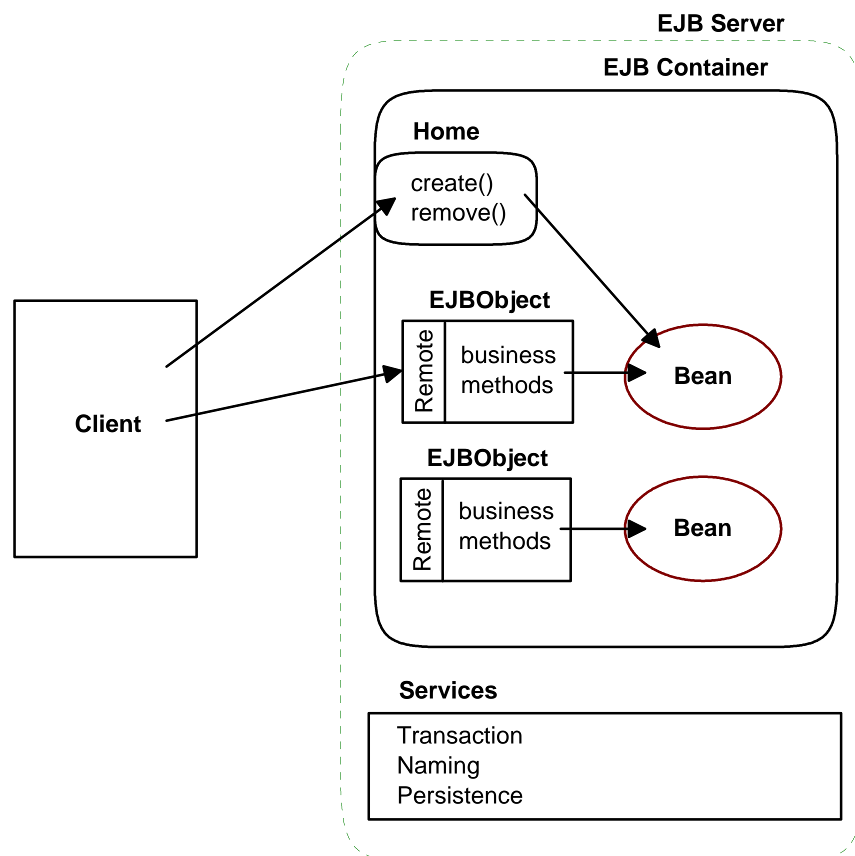
 Copyright © 1999 Nova Laboratories - Reproduction Prohibited

interface. The container vendor should provide the means to generate the home object implementation from the home interface.

The *remote interface* lists the business methods in the EJB class. The *EJBObject* implements the remote interface, and is the object that the client must use to access the business methods of the *EJB instance*. The EJB class developer defines the remote interface, and the container vendor should provide the means to generate the corresponding EJBObject.

The client never gets a reference to the EJB instance, only its EJBObject instance. When the client invokes a method, the EJBObject receives the request and delegates it to the EJB instance, providing any necessary wrapper functionality in the process.

The *client* is an application that uses the home object to locate, create, or destroy instances of an EJB class, and uses the EJBObject to invoke the business methods of an instance. The client can be written in Java™ and use Java™ RMI to access the home object and EJBObject, or it may be written in another language using CORBA®/IIOP™, providing the required server-side components are deployed in a manner that makes them accessible through CORBA® interfaces.



Here is an illustration of the architecture of Enterprise JavaBeans™. The next section will go into greater detail to describe the individual components.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

Benefits of Enterprise JavaBeans™

As a developer you should be considering whether Enterprise JavaBeans™ is appropriate for your application.

There are considerations when using EJB.

- the EJB server is a product that you must purchase
- you must implement a minimal API, and understand its semantics

The advantages far outweigh the disadvantages, especially for more complex applications.

Establishing Roles for Application Development

EJB allows the right person to do the right job. Business developers can focus on writing code that implements business functions. The deployer can take care of installation issues in a simple and portable fashion. The EJB server vendors will take care of providing support for complex services, and make them available to the bean often without the knowledge or assistance of the bean developer.

Automatic Transaction Management

One of these services transparently provided by the container vendor is transaction control.

The person writing the business functions does not have to worry about starting and terminating transactions, nor about whether or not the operations in this business function will work correctly in conjunction with business functions in other beans.

Distributed Transaction Support

Part of the transparency of transactions is through distributed transaction support. A client can, for example, begin a transaction and then invoke methods on beans in two different servers. Methods in one bean can call methods in another bean with the assurance that they will execute in the same transaction context.

Portability

With the exception of publicized vendor-specific enhancements, an enterprise bean will install and run in a portable fashion in any EJB server.

Scalability

While the architecture of EJB may appear to be somewhat complex, you'll find that the more you understand the architecture, the more you'll see that the Specification was written to allow vendors to provide extremely high-performance implementations. In addition, you should expect to see such features as load-balancing and failover once the EJB server market matures.

Integration with CORBA®

If you are familiar with CORBA®, a question may arise regarding the apparent competition between CORBA® and EJB. But in fact CORBA® and EJB are a

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

natural complement to each other. EJB is moving in the direction of using CORBA®/IIOP™ to provide a more robust transport mechanism. In addition, pure CORBA® clients will be able to access enterprise beans as EJB clients.

CORBAservices® provides a wealth of features to an application developer. Rather than trying to replace these services, EJB server vendors will likely include CORBAservices® into their product through a simplified API, allowing bean developers to use CORBAservices® without needing to become experts in CORBA®. An excellent example of this is the transaction support provided by EJB servers to enterprise beans. The transaction service is probably an implementation of CORBA® OTS. For an EJB server to work correctly with CORBA® clients, its transaction and naming services will have to support the CORBA® OTS and Naming Service interfaces, respectively.

Vendor Enhancements

The real value in Enterprise JavaBeans™ is the flexibility the specification allows for vendors to provide their own enhancements. Features like automatic object-relational mapping when using container-managed persistence, gateway services into existing applications, customizable business frameworks, and integration of CORBAservices® are just a few examples of added value that may be provided by vendors.

Roles in EJB Development

EJB Developer

The EJB developer writes EJB classes using the required classes and interfaces as defined in the Specification. Within this class are the methods for creating and removing the EJB instance, along with the business methods of this class.

The EJB developer is a Java™ developer with an understanding of the interfaces and semantics of Enterprise JavaBeans™, and understands the business needs that must be addressed. The EJB developer is responsible for implementing the functionality of the business application's server side as discrete business objects.

While the EJB container will usually handle all transaction control on behalf of the EJB instance, it is important that the EJB developer understand how transactions work so that he/she may stipulate to the EJB deployer the transactional needs of the various methods in the EJB class.

EJB Deployer

The EJB deployer is responsible for taking the EJB class and its supporting classes and installing them correctly in the EJB server. The deployer has an in-depth understanding of EJB and the characteristics of the runtime server environment, such as database type and location, and so forth.

The deployer receives the EJB class requirements from the EJB developer, such as transactional needs, names and descriptions of required environment properties, and so forth.

The deployer must make these properties, along with their correct runtime values, available to the EJB class at runtime. The deployer is also responsible

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

for ensuring that the home object for the EJB class is available in a namespace accessible through JNDI.

The EJB deployer may not be a Java™ developer nor understand the business rules that an EJB class implements, but understands the application framework in which the EJB instance runs.

For example, the deployer may not understand the details of a database application written using EJB, but knows the location, schema, and transactional requirements of the underlying database.The deployer may be the person to decide critical runtime attributes such as transaction isolation levels, and so forth. It is important for the EJB developer and deployer to communicate clearly to ensure that the EJB class is deployed with the correct deployment attributes.

EJB Container Vendor

The EJB container vendor provides software to install an EJB class and its supporting classes into an EJB server. The container vendor must also supply runtime classes that can provide the required services to the EJB instances at runtime. Examples of additional tasks are the generation of stub and skeleton classes to provide access to the EJB class' home object and EJBObjects, installation of references to the home object in a JNDI-accessible namespace, and availability of a suitable EJBObject implementation that can provide correct proxy services for the EJB class.

EJB Server Vendor

The EJB server vendor provides an application framework in which to run EJB containers.

The EJB server will implement, or provide access to, required services such as a JNDI-accessible naming service and an OTS-compatible transaction service.

Since there is currently no standard interface between the EJB server and the EJB container, the EJB container is actually provided by the EJB server vendor.

Application Developer

The application developer writes the client applications that use EJB classes. A client may be a Java™ applet or application, a servlet, or a natively compiled application that uses a CORBA®/IIOP™ interface to access the EJB components.

The application developer is providing a business application that uses the available EJB classes and methods as abstractions for obtaining low-level application services. This allows the application developer to concentrate on high-level functions such as data representation to the user without having to worry about how such data is obtained.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

3 - Understanding the EJB Components

You should now be familiar with the overall architecture of EJB, along with its major components. This section will describe these components in considerably more detail, and will explain their runtime behavioral semantics.

Home Interface

Factory for Enterprise Beans

A client needing the use of an enterprise bean creates one through its home interface. The home interface lists one or more `create()` methods that can be used to create an instance of this enterprise bean. This home interface is not implemented by the bean, but by some other class we'll call the ***home object***. An instance of this home object is instantiated within the server and made available to clients as a factory for the enterprise bean.

Locating the Home Object

A reference to the home object is placed in a naming service that is accessible from the client using JNDI. The EJB server will likely provide some sort of namespace implementation, although an external namespace could be used. In either case the location of the namespace and the JNDI context factory class name must be provided to the client initially. For example, a client applet might receive the namespace location and JNDI context factory class name as applet parameters.

In addition to providing the location and class name, the client must also have some knowledge of how to locate the home object within the naming tree. This should also be provided to the client at startup. When the deployer installs the enterprise bean into the EJB server, he/she may have the option of specifying a particular location in the naming tree, such as

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

ejb/accounting/AccountsPayable. The client must be given this fully-qualified pathname to locate and obtain the reference to the AccountsPayable home object.

It is incorrect to say that a container is made available to clients through JNDI. Clients look up a home interface implementation using JNDI. That implementation may be provided by a custom container, but this is a vendor-specific detail that should be ignored by both the enterprise bean developer and the client.

Methods in the Home Interface

The developer that defines ejbCreate() methods in the enterprise bean must also declare corresponding create() methods with matching signatures in the enterprise bean's home interface.

Entity beans may also have *finder methods* which allow clients to locate existing entity beans based on their identity.

The home interface is defined as extending from javax.ejb.EJBHome. This interface has the following methods:

```
public interface javax.ejb.EJBHome extends Remote {
  public EJBMetaData getEJBMetaData() throws
                                  RemoteException;
  public void remove(Handle handle) throws
                                  RemoteException,
                                  RemoveException;
  public void remove(Object primaryKey) throws
                                  RemoteException,
                                  RemoveException;
}
```

The home interface for a bean might look like this:

```
public interface myHome extends EJBHome {
  public myRem create() throws    RemoteException,
                                  CreateException;
  public myRem create(String str) throws
                                  RemoteException,
                                  CreateException;
}
```

where

```
public interface myRem extends EJBObject { ... }
```

The container vendor is responsible for providing the home object which implements this home interface, since only the vendor can implement the code that can act as the factory to create the enterprise beans.

 Copyright © 1999 Nova Laboratories - Reproduction Prohibited

Container

Defining the Container

In reading the Enterprise JavaBeans™ 1.0 Specification the term *container* should not be taken literally as a class, but as a level of responsibility where a set of services must be performed on behalf of the bean. A container vendor will provide a set of tools and classes that run within an EJB server that fulfill these responsibilities.

Some of these services are:

- swapping to and from secondary storage (for session beans)
- persistence management (for entity beans)
- availability of a home object implementing creation and lookup services
- visibility of the home object in a JNDI-accessible namespace
- proper creation, initialization, and removal of beans
- ensuring that business methods run in the proper transaction context
- implementation of certain basic security services
- generation of stubs and skeletons for remote method invocation on the home object and EJBObject

Container and EJBObject as a Pair

The Specification often refers to services that may be provided by either the container or the EJBObject. These references are illustrative and do not imply service requirements of a specific class.

Both the EJBObject and the container class(es) are provided by the container vendor to support an enterprise bean. Together these classes must fulfill the responsibilities of the bean container.

Both the container and the EJBObject have distinct entry points into the bean, giving each a unique ability to provide support for a specific service. For example, the container knows which transaction attributes apply to the methods of the bean by reading its deployment descriptor. However, it is through the EJBObject that these business methods are invoked.

The EJBObject must communicate with the container to determine in what transaction context to call the business method. Once this is determined, the EJBObject must establish this transaction context before invoking the business method.

All that matters is that the EJBObject and container work together to implement the services required by a container. The container vendor provides the implementations of both objects, and is free to partition this work any way it likes.

Relationship to the Home Interface

Vendors currently provide their own tools which read the home interface and generate the home objects as containers. In this case the vendor is using a separate container class for each enterprise bean class.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

A container vendor is free to use some other implementation strategy, such as one container class that implements multiple home interfaces, or even a standard container class with separately created custom home implementation objects. The only requirement is that the container vendor make a home object available to clients through JNDI.

Neither the client nor the bean developer are concerned with the implementation details of the container and home object.

Enterprise JavaBean

The enterprise bean is the class that the developer writes to provide application functionality.

The developer has the option of creating either a session bean or an entity bean, and will make this distinction by implementing the appropriate interface and declaring its type in the deployment descriptor.

For session beans:

```
public class myBean implements javax.ejb.SessionBean ...
```

For entity beans:

```
public class myBean implements javax.ejb.EntityBean ...
```

The methods in your enterprise bean will never be invoked directly from the client. The client calls the bean's methods indirectly through the EJBObject, which acts as a proxy. In routing all invocations through the EJBObject, the container vendor can insert its own functionality into the invocation through wrapper code, or *method interposition*. An example of this would be to create a new transaction context for each method invocation, or to commit or roll back the transaction when the method returns (to the EJBObject).

When the container vendor's tool generates stubs and skeletons as part of the installation process of a bean, it creates a stub and skeleton for the bean's EJBObject. It doesn't actually create a stub or skeleton for the bean itself, since it is never accessed over the network. The EJBObject is the true network object. The bean is the delegate that contains the application-specific business code.

The container may also call certain methods in the bean. For example, the container will ensure that after a new instance of a bean is created, that the arguments used in the call to `create()` on the home object are passed to the corresponding `ejbCreate()` in the bean.

There are additional interfaces and requirements for enterprise beans. However, these requirements vary based on whether the bean is a session bean or an entity bean. These will be covered in the subsequent sections detailing session and entity beans.

Remote Interface

After writing the enterprise bean the developer created a home interface to specify the signatures of the creation methods that should be made available to the client, with the restriction that for each `create()` method specified in the home interface, there had to be a corresponding `ejbCreate()` method in the bean.

 Copyright © 1999 Nova Laboratories - Reproduction Prohibited

In a similar fashion, the developer must create a remote interface which describes the business methods of the bean that the developer would like the client to be able to invoke.

Since all client invocations come through the EJBObject, it is the EJBObject, not the home object, that will formally implement this interface.

The method names and signatures listed in the remote interface exactly match the method names and signatures implemented in the bean. This differs from the home interface, whose method signatures matched, but the names were different.

Here is an example of a remote interface:

```
public interface Account extends javax.ejb.EJBObject {
  public void deposit(double amount)  throws RemoteException;
  public void withdraw(double amount) throws RemoteException;
  public double balance()             throws RemoteException;
}
```

The methods all declare that they may throw a `RemoteException`, since the specification states that the client-side stub is RMI-compliant. Keep in mind that this does not preclude the stub/skeleton implementation from using a different transport, such as CORBA®/IIOP™.

The remote interface extends the `javax.ejb.EJBObject` interface, which adds additional method requirements.

EJBObject

The EJBObject is the network-visible object, with a stub and skeleton, that acts as a proxy for the bean. The bean's remote interface extends the EJBObject interface, and the EJBObject class implements this remote interface, making the EJBObject class specific to the bean class. For each bean class there will be a custom EJBObject class.

Here is the definition of the EJBObject interface, which the bean's remote interface extends:

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {
  public EJBHome getEJBHome()      throws RemoteException;
  public Object  getPrimaryKey()   throws RemoteException;
  public Handle  getHandle()       throws RemoteException;
  public void    remove()          throws RemoteException,
                                          RemoveException;
  public boolean isIdentical(EJBObject other) throws
                                         RemoteException;
}
```

The EJBObject class that implements this interface is an RMI server object, since it is implementing an RMI remote interface. Note that the bean itself is not a remote object and is not visible over the network. When the container instantiates this EJBObject class it will initialize it with the reference to the bean instance, so that it may delegate the business method calls appropriately.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

The vendor implementation will likely maintain this one-to-one relationship between an EJBObject instance and a bean instance.

Since the remote interface includes the methods of the EJBObject interface, the bean should obviously not implement this interface, although it does provide implementations of the listed business methods.

Since the EJBObject must formally implement the remote interface of the bean, the container vendors generate the EJBObject's source code at bean installation time, where the generated code implements the bean's remote interface. The EJBObject is typically given a unique class name and associated with the bean as its EJBObject class.

Session Bean

A session bean is an enterprise bean where each instance of a session bean is created through its home interface and is private to that client connection. The session bean instance cannot be easily shared with other clients. This allows the session bean to maintain the client's state.

An example of this might be a shopping cart bean, where many customers can shop simultaneously, adding to their private cart, instead of everyone adding their personal items to a community shopping cart.

Defining a Session Bean

You create a session bean by defining a class that implements the `javax.ejb.SessionBean` interface. The interface is defined as follows:

```
public interface javax.ejb.SessionBean extends
                      javax.ejb.EnterpriseBean {
  public void ejbActivate()        throws RemoteException;
  public void ejbPassivate()       throws RemoteException;
  public void ejbRemove()          throws RemoteException;
  public void setSessionContext(SessionContext context)
                                   throws RemoteException;
}
```

The `javax.ejb.EnterpriseBean` is an empty interface, and is the supertype for both session and entity beans.

Swapping Session Beans

The container vendor may optionally implement a swapping mechanism to move instances of session beans from memory to secondary storage, increasing the total number of beans that can appear to be instantiated at a time. The implementation would presumably maintain a timeout threshold on its beans, and when a bean's inactivity period has reached the threshold, it would be copied to secondary storage and deleted from memory.

A container can use any mechanism it chooses to store the bean persistently. The most likely means of doing this would be serialization of the bean. The bean developer should avoid using `transient` fields in the bean. Instead, the `ejbActivate()` and `ejbPassivate()` should be used to maintain the fields values.

 Copyright © 1999 Nova Laboratories - Reproduction Prohibited

Activation and Passivation

To provide support for swapping by the vendors, the specification formally defines passivation as the process of swapping out a bean, and activation as the process of restoring it to memory. The methods `ejbActivate()` and `ejbPassivate()`, declared in the `SessionBean` interface, allow the container to tell the bean that it is about to be swapped out, and that it has just been swapped in. The bean developer can use these methods to release and restore values, references, and system resources that should not be held while the bean is in a passivated state. An example of this might be a database connection, since connections are limited resources and cannot be used by a bean while it is passivated.

Using these methods makes the use of `transient` unnecessary. In fact, using `transient` can be somewhat unsafe, since it results in fields that are reset to null or 0 implicitly by the serialization mechanism. It is better that the developer explicitly maintains these fields through the `ejbActivate()` and `ejbPassivate()` methods.

Relying on Java™ serialization to set `transient` fields to null is also non-portable since this behavior would change if this bean were deployed in an EJB container which did not use Java™ serialization to achieve persistence.

If the container does not provide swapping, then these methods will never be called.

The passivated bean is activated as a result of a business method call from the client. When the EJBObject receives the method invocation request it informs the container that the bean must be activated. Once the activation is complete the EJBObject delegates the method call to the bean.

If the bean is currently in a transaction, it will not be passivated. It is more efficient to leave the bean in memory, since transactions normally complete within a fairly short period of time.

If the bean does not have state that must be released prior to passivation or reset after activation, then these methods can be left empty. In most cases, the bean developer should not expect to have to do anything in these methods.

Session Bean Management State

A session bean's deployment descriptor must declare the bean as either *stateless* or *stateful*. A stateless bean is one which does not maintain any state information between method calls. In general, the benefit of a session bean is that it does maintain state on behalf of the client.

However, there is a benefit to having session beans that are stateless. Stateless session beans are not swapped (or passivated). Since they do not maintain state, there is nothing to preserve. The container instead has the option of destroying the bean instance. The client is never aware that the bean has been destroyed. The client's reference is to the EJBObject. If the client later invokes a business method, the EJBObject will work with the container to instantiate a new session bean. Since there is no state, there is nothing else to restore.

Also, stateless session beans can be shared by clients, with the restriction that only one client may be executing a method within the bean at a time. Since there is no state maintained between the method calls, any client can use any

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

stateless bean instance. This would allow the container to possibly maintain a smaller pool of reusable session beans, saving memory.

Since a stateless session bean cannot maintain state between method calls, it is technically an error to define `create()` methods in the home interface that take arguments. Passing arguments to the bean during creation implies that state may be maintained in the bean after the call to `ejbCreate()` returns. Furthermore, the container must be able to fully recreate a stateless session bean as a result of a business method call on the EJBObject. At that point it would no longer have the startup arguments that were used to create the bean initially. The vendor's installation tool should check that the home interface of a stateless session bean does not contain `create()` methods with arguments.

Entity Beans

Role of the Entity Bean

Entity beans are used to represent underlying objects. The most common application for entity beans is their representation of data in a relational database. A simple entity bean can be defined to represent a row in a database table, where each instance of the bean represents a specific row. More complex entity beans could represent views of joined tables in a database, where one instance represents a specific customer and all of that customer's orders and order items. For these beans there should be considerable demand for vendor enhancements, such as integrated object-relational mapping.

In most cases it is both simpler and more efficient to have an entity class represent one database table than a set of related tables. Relationship traversal can be easily added to the entity class definitions, thus maximizing cache reuse and minimizing the presence of stale data.

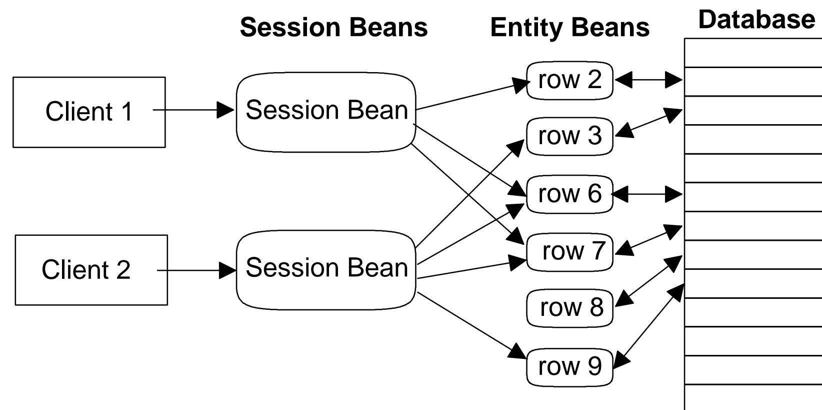
Comparing Entity and Session Beans

It may seem that session beans are not very useful, especially for applications that are database-driven. This is certainly not true. Since an entity bean instance represents (for example) an underlying row in a database, the relationship between the entity bean instance and the database row is exactly one-to-one. Since multiple clients must be able to access that underlying row, it means that, unlike session beans, the entity bean instances are shared among clients. Since they are shared, they do not allow storage of per-client state information.

Session beans allow the client to store state information on behalf of the client, and the relationship between the client and the session bean instance is one-to-one. Entity beans allow storage of row information, making the relationship between the entity bean instance and the row one-to-one. An ideal solution might be a client which calls into the server through a session bean, and the session bean accesses the database through entity beans. This allows for storage of state information both for the client and for the rows in the database.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

The following diagram illustrates this:



Session beans are also invaluable for providing overall control of transactions across invocations of methods in the same or different EJB classes. Without session beans it might be necessary for the application developer (i.e., client developer) to understand the transactional requirements of the EJB classes, and to utilize *client-demarcated transactions* to provide transaction control. One major benefit of EJB is that an application developer can write applications without requiring knowledge of the transactional requirements of the EJB classes. A session bean can be created to represent a business operation, and that session bean can control the transaction, making client-demarcated transactions unnecessary.

Finder Methods

Creating and removing bean instances, either through the home or remote interface, have different meanings for entity beans than session beans. For session beans, *removing* means the bean instance is removed from the container and cannot be used again, and its state information is lost. For entity beans, removal means that the underlying row in the database is deleted. For this reason, removal is not considered a part of the typical lifecycle of the entity bean.

The only way to have a session been is to create it. Creating an entity bean means that a row is inserted into the database. Similar to the removal operation, the creation operation is generally not considered part of the typical lifecycle of an entity bean.

For a client to get access to an entity bean, it *finds* it. In addition to `create()` methods, the home interface of an entity bean will also have *finder methods.* The client must determine some means of identifying the specific row based on some application qualification. For example:

```
public interface AccountHome extends EJBHome {
  public Account findByFirstLast(String first, String last)
                 throws RemoteException, FinderException;
  public Account findByAccountNumber(String acctNum)
                 throws RemoteException, FinderException;
}
```

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

When the client invokes one of these methods on the home object, the container passes the invocation to the corresponding method in the entity bean.

```
public class myEntityBean implements EntityBean {
  ...
  public Object ejbFindByFirstLast(String first, String last){
         // runs appropriate singleton SELECT statement
         // returns primary key for selected row
  }
  public Object ejbFindByAccountNumber(String acctNum) {
         // runs appropriate singleton SELECT statement
         // returns primary key for selected row
  }
}
```

A good way to think of each finder method is as a database SELECT statement, where the dynamic SQL parameters are supplied as the method arguments.

Notice that the finder methods in the home interface return a remote reference (to the EJBObject) back to the client. The finder methods in the bean return a unique identifier, or *primary key*, to the container. The container will take this primary key and use it to instantiate a new EJBObject to represent the selected row. Regardless of the implementation of the finder method, the selected row is always represented to the container using a primary key. It is up to the entity bean class to decide how to construct a primary key to uniquely identify the row.

It is likely that a finder method may find several rows that meet the criteria in the SELECT statement. In this case the bean's finder method is defined to return an enumeration of primary keys. The finder method in the home interface is defined to return to the client an enumeration of EJBObject references.

```
public interface AccountHome extends EJBHome {
  ...
  public Enumeration findByCompany(String companyName)
                                 throws RemoteException,
                                        FinderException;
}
public class myEntityBean implements EntityBean {
  ...
  public Enumeration ejbFindByCompany(String companyName) {
         // runs appropriate SELECT statement
         // returns an Enumeration of primary keys
  }
}
```

Primary Key

The term *primary key* can be misleading. It would be more appropriate to consider it a *unique identifier*. In the case of an entity bean that represents a

 Copyright © 1999 Nova Laboratories - Reproduction Prohibited

row in a table, the primary key might be a representation of the values of the composite key for that row.

For every entity bean instance, there is a corresponding EJBObject. When an EJBObject is paired with an entity bean instance, the primary key of the instance is stored in its EJBObject.

This entity bean instance is said to now have an *identity*.

When the client invokes a finder method on the home object, the container will use an entity bean instance that has no identity to perform the request. The container may privately hold one or more of these anonymous instances for this purpose. Regardless of the implementation of the finder method, its implementation in the bean will always return to the container just the primary key of the underlying data, such as the row in the database. If multiple rows met the search criteria, then multiple primary keys are returned to the container.

Now that the container has the primary key (or keys) it will instantiate an EJBObject (or EJBObjects) and initialize them with the primary keys. The container then has the option of also instantiating the entity bean instances to associate with each EJBObject. Since the identity of the underlying row is actually in the EJBObject, there would be no state within the bean instances. For this reason, the container may defer the instantiation of the bean instances until a business method request is made on the EJBObject, conserving memory resources.

When a finder method returns a primary key to the container, the container will first check to see if an EJBObject with that primary key already exists. If so, the container will not create a new EJBObject, but will return the reference to the existing EJBObject to the client. This assures that there is only one EJBObject instance per row, and that all clients share the EJBObjects.

The primary key uniquely identifies the bean instance only within its class, or home. The container is responsible for ensuring this distinction.

It is important to remember that the purpose of the finder methods is to extract only primary key data from the database. The actual column data for the row is not extracted. It is likely that no entity bean instances are allocated as a result of the call to a finder method. Only the EJBObjects containing the primary keys are allocated. The entity bean instances will be allocated and loaded at a later time as a result of a client's method call on the EJBObject.

The home object will always make the following method available to clients:

```
public myRem findByPrimaryKey(Object key) throws ...;
```

The EJBObject provides an implementation of the following method:

```
public Object getPrimaryKey();
```

The client can obtain the primary key of an entity at any time, and use the primary key at a later point in time to re-establish a reference to the entity through its home interface.

The class type of the primary key is specified in the deployment descriptor. The bean developer may represent the primary key using any class type. The only requirement is that the class must implement `Serializable`, since it must be possible to pass the primary key between the client and server.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

Swapping Entity Beans

Now would be a good time to look at the `javax.ejb.EntityBean` interface.

```
public interface javax.ejb.EntityBean extends EnterpriseBean {
  public void ejbActivate()         throws RemoteException;
  public void ejbPassivate()        throws RemoteException;
  public void ejbRemove()           throws RemoteException,
                                           RemoveException;
  public void setEntityContext(EntityContext ctx)
                                    throws RemoteException;
  public void unsetEntityContext()  throws RemoteException;
  public void ejbLoad()             throws RemoteException;
  public void ejbStore()            throws RemoteException;
}
```

Activation and passivation work in a similar fashion to session beans. However, entity beans are stateless when they are not in a transaction; their state is always synchronized with the underlying data store. If we apply the same rules for swapping that we used for session beans, then we would not swap out a stateless entity bean, we would just destroy it. However, since the container needs anonymous entity beans to invoke the finder methods, the container will likely passivate entity beans into a private in-memory pool for this purpose. Once the entity bean instance is taken away from the EJBObject, it no longer has an identity (or primary key association).

The container may also use this in-memory pool for re-allocation of entity beans when a business method is requested on an EJBObject that has no corresponding entity bean.

Remember that when a bean is in this pool it has no identity and can be reused by any other EJBObject. The container may implement a policy which does not maintain any entity beans with EJBObjects, except when a business method is currently be executed through that EJBObject. More properly, the entity bean would remain associated with the EJBObject while the bean is involved in a transaction.

Bean-Managed Persistence

Since an entity bean represents underlying data, we need to fetch the data from the database and place it in the bean. When the container first associates an entity bean instance with an EJBObject, it starts a transaction and invokes the `ejbLoad()` method of the bean. Within this method the developer must provide the code that will extract the appropriate data from the database and place it in the bean. When the container wishes to commit the transaction it will first invoke the bean's `ejbStore()` method. This method is responsible for writing the data back to the database.

We call this bean-managed persistence, since is the code within the bean's methods that provide this synchronization.

When the `ejbLoad()` method completes, there is the potential for the bean to become out of synch with the underlying database. The business method invocation which triggered the allocation of the bean to the EJBObject, and subsequent `ejbLoad()`, must be declared in the deployment descriptor to run

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

within a transaction. Upon receiving the business invocation request, the EJBObject will work with the container to establish a transaction context. The container will then allocate the bean to the EJBObject and invoke the bean's `ejbLoad()` method. This method is now running in the context of a transaction. This transaction context is passed to the database, where the accessed rows in the database become locked by the transaction, according to the transaction isolation level specified in the deployment descriptor.

The rows in the database remain locked as long as the transaction context is alive. When the transaction is about to be committed, either by the client or the container, the container will first invoke the bean's `ejbStore()` method, causing the bean to flush its data to the database.

Keeping the appropriate database rows locked between the `ejbLoad()` and `ejbStore()` ensure that the bean is always synchronized with the database. Multiple business method invocations could occur during this time. More specifically, the `ejbLoad()` and `ejbStore()` demarcate the transaction boundaries, and multiple business method invocations can occur within the transaction.

The duration of the transaction is determined by the deployment descriptor, and possibly by the client.

Note that the `ejbActivate()` and `ejbPassivate()` methods should not be used to perform synchronization with the database.

Container-Managed Persistence

If the deployment descriptor declares that the bean will utilize container-managed persistence, the `ejbLoad()` and `ejbStore()` methods will not be used to access the database. The container will load the data from the database into the bean and then invoke the bean's `ejbLoad()` method to tell it that it has just received data from the database.

Likewise, the container will invoke the bean's `ejbStore()` method to tell it that its data is about to be written to the database. These methods do not actually perform any database operations.

While a vendor may provide sophisticated tools to accomplish this, such as automatic generation of entity bean classes utilizing object-relational mapping, the specification describes a minimal set of requirements for a vendor to provide container-managed persistence.

The deployment descriptor can specify a simple mapping of `public` fields in the bean to columns in the database. The container uses this to read the public fields from the bean and write them into the appropriate columns, and to read the database columns and write them into the public fields.

While container-managed persistence can be a great service to EJB developers, without having a more sophisticated mechanism, such as object-relational mapping, the developer may find it more effective to use bean-managed persistence.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

Deployment Descriptor

Distinguishing EJB Development Roles

There are two primary roles for EJB development, the bean developer and the bean deployer.

There are a number of attributes which cannot be predicted by the developer, such as network addresses of databases, database drivers to use, and so forth. The deployment descriptor acts as property sheet which can defined by the developer, and filled in with correct values by the deployer. The deployment descriptor is a standard format, and so is portable between the development and deployment environments, even when different EJB platforms are used.

Controlling the Behavior of Enterprise Beans

In addition to providing a standard property sheet for collaboration between development and deployment time, the deployment descriptor also contains detailed information about how the bean should execute regarding transactions and security. Certain attributes, such as access control lists, will likely be adjusted by the deployer to ensure that the appropriate users can utilize the beans at runtime. Other attributes, such as transaction information, will probably be specified completely by the developer, since it is the developer who created the database access code and has intimate knowledge of how the bean's methods should run regarding transactions.

Defining the Deployment Descriptor

The deployment descriptor is a standard Java™ class. An instance is created, populated with values, and serialized. This serialized deployment descriptor is placed in a jar file and sent to the deployment area along with the enterprise bean classes. The deployer will read in the serialized deployment descriptor, possibly modify some values, and use this modified deployment descriptor to install the enterprise bean.

Here are the contents of the deployment descriptor. It is the superclass to two other deployment descriptor classes. The subclasses are the descriptors actually used to describe the bean.

`javax.ejb.deployment.DeploymentDescriptor`

- bean home name
- bean class name
- home interface class name
- remote interface class name
- environment properties
- control descriptors
- access control list

These two classes are subclasses of `DeploymentDescriptor`:

`javax.ejb.deployment.SessionDescriptor`

- state management type
- session timeout

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

`javax.ejb.deployment.EntityDescriptor`

- list of container-managed fields
- primary key class name

This descriptor defines transaction and security attributes for the entire bean and for individual methods. An array of these objects are specified in `DeploymentDescriptor`.

`javax.ejb.deployment.ControlDescriptor`

- transaction isolation level
- `Method` object to which this descriptor applies
- run-as mode (for identity mapping)
- run-as identity (for identity mapping)
- transaction attribute

To deploy an enterprise bean the appropriate descriptor is allocated, initialized, and serialized, and placed in the jar file along with the enterprise bean classes.

Each vendor may have a different means for defining a deployment descriptor. One may use a text specification file, for instance, while another provides a graphical tool. The end result is a descriptor which is in a standard format, and is portable across vendor platforms.

EJB Jar File

To package the enterprise bean, the bean classes, including the interfaces, and the serialized deployment descriptor are placed in a jar file. This jar file must have a manifest file entry which declares the deployment descriptor as an enterprise bean.

```
Name: AccountDD.ser
Enterprise-Bean: true
```

It is the deployment descriptor, not the bean class, that is listed in the manifest as an enterprise bean. The deployment descriptor provides the full description of all the files in the jar that together define the enterprise bean.

The developer should not have to be concerned with the creation of the EJB jar file. The vendor should provide a tool which will assist the developer in creating the deployment descriptor, and then package all the necessary files into a jar with the appropriate manifest file entries.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

This page intentionally left blank

 Copyright © 1999 Nova Laboratories - Reproduction Prohibited

4 - Transactions

CORBA® OTS

The transaction model used in EJB similar to OTS. In fact, CORBA®-compliant EJB servers must provide an OTS-compliant transaction service. Understanding how OTS works helps to understand how transactions work in EJB.

Defining Transactions

A transaction is formally defined as an atomic unit of work. Multiple operations can be included in a transaction, and when the transaction is terminated, all changes performed by the operations are either applied or undone as a whole. These operations are called `commit` and `rollback`.

Transactions are used extensively with database applications. Good database products provide very strong support for transactions. Rows that are accessed during a specific transaction become locked and stay locked for the duration of the transaction. Based on the database product there can be multiple locking levels which can be selected prior to the start of the transaction. The appropriate locking level should guarantee integrity of the data while optimizing concurrency of access for operations in other transactions.

Transactions may also be distributed across the network, such as a client accessing two different databases from within a single transaction. To support distributed transactions most transaction managers (including databases) support two-phase commits. This is a simple protocol where the transaction manager first asks all participants if their work is finished and would like to attempt to commit their work. This is the first phase of the protocol. Once everyone agrees to commit, the second phase begins. The transaction manager gives the command to perform their individual commit operations.

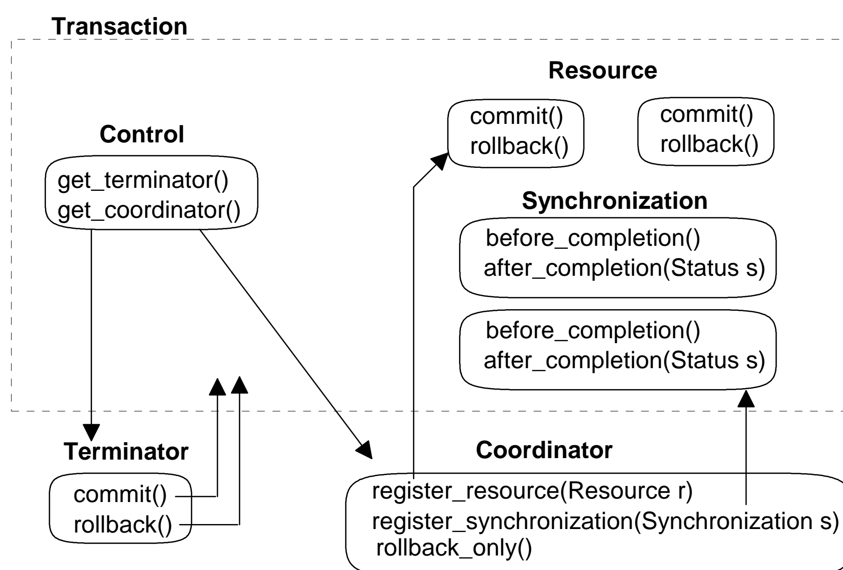
Copyright © 1999 Nova Laboratories - Reproduction Prohibited

Key Components in OTS

To better understand how OTS works we need to look at its key components. The following components can be mapped almost directly to EJB, and understanding how these components work in OTS will provide a good foundation for understanding transactions in EJB.

- Control
- Terminator
- Coordinator
- Resource
- Synchronization

The following diagram illustrates the significant methods defined for these objects, and how the objects fit into the transaction architecture.



The dashed box represents a transaction. Within the transaction are all of the objects participating in the transaction. Commits and rollbacks will be applied to all the `Resource` objects in this group.

The `Control` object represents the transaction. From this object we can obtain the `Coordinator` and the `Terminator`. An EJB developer never sees the `Control` object. It is used by the container to manage the transaction on behalf of the bean.

The `Terminator` is also used by the container to commit or roll back the transaction when a thread returns from a bean method where the method is described in the deployment descriptor as requiring the container to terminate the transaction upon completion of the method. When a commit or rollback is requested, all the objects in the transaction will be committed or rolled back, as appropriate.

The `Resource` is the object that has transactional state. For example, it may be a connection to a database. Calling `commit()` on this object would cause the database updates to be applied to the database. A `rollback()` would revert all changed made to the database through this connection since the start of the

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

transaction. Once the commit or rollback completed, the corresponding rows in the database would be unlocked. The level of locking applied would have been specified in the deployment descriptor. The full set of methods for this object would show that these objects actually implement the two-phase commit protocol, allowing each object to vote as to whether the entire transaction should be committed or rolled back.

The `Synchronization` is an object that wishes to be notified upon completion of the transaction whether the transaction was committed or rolled back. Unlike the `Resource`, it does not participate in the two-phase commit protocol and has no vote as to whether the transaction should be committed or rolled back. It plays a passive role in the transaction. This is the role that a session bean may play by implementing a special interface.

The `Coordinator` is the object that makes all this work. Through this object both `Resource` objects and `Synchronization` objects are registered in the transaction. The bean does not get access to this object directly. Transaction-aware objects that are intended for use with EJB will transparently obtain a reference to the current transaction's `Coordinator` to register itself.

Transactional vs Recoverable Objects

OTS distinguishes transactional and recoverable objects. This distinction is relevant to EJB. While the full definition of these types is rather extensive in the CORBAservices® OTS Specification, it essentially means that recoverable objects have a `commit()` and `rollback()` method, allowing the transaction to directly manipulate their state or behavior. A transactional object does not have these methods, and cannot be influenced by the transaction. However, a transactional object is an object that has a transaction associated with it, so that recoverable objects (or `Resources`) that are allocated will be associated with the transactional object's current transaction.

An enterprise bean is a good example of a transactional object. The container will maintain a transaction on behalf of the bean. Any recoverable objects that are allocated by the bean are transparently placed into the current transaction with the help of the container. The bean does not have a `commit()` or `rollback()` method, and so the transaction cannot manipulate the bean directly. It doesn't really make sense to make a bean a recoverable `Resource` since this would require additional work on the part of the bean developer for every bean written, and enterprise beans rarely have internal state that should directly affect the outcome of a transaction. They function better as managers of recoverable objects, letting the recoverable objects do the work.

Note that a bean can vote to roll back a transaction prior to the container attempting a commit or rollback. The `rollback_only()` method in the `Coordinator` is provided to the bean in the `EJBContext` as `setRollbackOnly()` so that the bean can flag the transaction as requiring a rollback when the time comes to terminate the transaction.

A bean can still be told of the outcome of a transaction through the `SessionSynchronization` interface.

Specifying Transaction Controls in the Deployment Descriptor

The deployment descriptor for a bean contains an array of `ControlDescriptor` objects. Each `ControlDescriptor` describes, among

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

other things, the transaction control that should be associated with that method.

The bean developer specifies the transaction controls for the methods in the bean. The deployer generally should not change these values without detailed knowledge of the transaction-related behavior of the method.

The following six transaction controls are integer constants defined in the `ControlDescriptor` class. Aside from the methods of this class, there are no other APIs that access this information. The bean itself does not get access to its transaction controls for its methods.

The container will read these control values to maintain the appropriate transactional behavior for the bean.

- `TX_NOT_SUPPORTED`
- `TX_SUPPORTS`
- `TX_REQUIRED`
- `TX_REQUIRES_NEW`
- `TX_MANDATORY`
- `TX_BEAN_MANAGED`

You would apply the appropriate value to the `ControlDescriptor` for your bean's method typically through a vendor-provided tool for creating deployment descriptors.

TX_NOT_SUPPORTED

This method should not run in a transactional context. If the executing thread is running within the transaction, the transaction will be suspended until the thread returns from the method.

TX_SUPPORTS

While a transaction is not required for this method, the thread may have an active transaction when running this method.

TX_REQUIRED

This method must be run within in transaction. If the thread already has a transaction, the thread is allowed to enter the method. If the thread does not have a transaction, the container will start one on behalf of the thread, allow the thread to enter, and terminate the transaction once the thread returns. Normally the transaction is committed. If the thread invoked the `setRollbackOnly()` method, the container will perform a rollback instead.

TX_REQUIRES_NEW

Regardless of whether or not the thread has a transaction, the container will create a new transaction for the duration of this method. When the thread returns, the container will either commit or roll back the transaction. If the thread already had a transaction underway, it would be suspended until after the thread returned and the method's transaction was terminated.

 Copyright © 1999 Nova Laboratories - Reproduction Prohibited

TX_MANDATORY

The thread must already be in a transaction when attempting to call this method. If the thread does not have a transaction, the container will throw an exception.

TX_BEAN_MANAGED

This control is somewhat different from the previous controls. A method with this control is stating that the container should not play any part in transaction management on the part of the method. Instead, the method is going to create and terminate its own transaction through a special interface that is accessible only for methods with this transaction control. This control will be covered in more detail in a later section.

JTS - Java™ Transaction Service

The JTS is actually not a transaction service, but an interface to an underlying transaction service provider. JTS is very simple, consisting of one interface an several exceptions. It is apparent by looking at its list of exceptions that it was modeled to work with OTS, although it could be used as an interface to other transaction services also.

For beans that declare their transaction control as bean-managed, the bean can get access to the transaction service through this interface. It can also be used for vendors that provide support for client-demarcated transactions.

Here is the interface definition for `UserTransaction`:

```
public interface javax.jts.UserTransaction {
  public void begin()  throws
                             IllegalStateException;
  public void commit() throws
                             TransactionRolledBackException,
                             HeuristicMixedException,
                             HeuristicRollbackException,
                             SecurityException,
                             IllegalStateException;
  public void rollback() throws
                             SecurityException,
                             IllegalStateException;
  public void setRollbackOnly() throws
                             IllegalStateException;
  public void setTransactionTimeout(int seconds);
  public int getStatus();

  // STATUS_ACTIVE, STATUS_COMMITTING,
  // STATUS_COMMITTED, STATUS_MARKED_ROLLBACK
  // STATUS_NO_TRANSACTION, STATUS_PREPARED
  // STATUS_PREPARING, STATUS_ROLLEDBACK
  // STATUS_ROLLING_BACK, STATUS_UNKNOWN
```

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

```
}
```

Bean-Managed Transactions

A bean can get access to the transaction service provided the bean is declared with a transaction control of TX_BEAN_MANAGED. While transaction controls apply to individual methods, this control can only be applied to the entire bean. The ability of the bean to access the transaction service cannot be selectively applied to only certain methods of the bean.

Therefore, it is an error for one method of the bean to declare the transaction control of TX_BEAN_MANAGED while another method declares a different transaction control. The vendor's installation tool should detect this and report an error.

The bean gets access to the transaction service through the SessionContext or EntityContext provided to bean initialization as the argument to setSessionContext() or setEntityContext(), respectively. Each of these interfaces subclasses from EJBContext. The definition of EJBContext is as follows:

```
public interface javax.ejb.EJBContext {
  public Identity   getCallerIdentity();
  public boolean    isCallerInRole(Identity other);
  public EJBHome    getEJBHome();
  public Properties getEnvironment();
  public UserTransaction getUserTransaction() throws
                                  IllegalStateException;
  public boolean    getRollbackOnly();
  public void       setRollbackOnly();
}
```

Once the bean obtains a UserTransaction reference, it can use the reference to manage its own transactions.

For stateful session beans, a method in the bean can create a transaction and return without terminating the transaction. Upon a subsequent call to one of the methods of the bean, the container will detect that a bean-created transaction is still in effect, and if the transaction being carried by the calling thread is the same transaction, the container will allow the thread to re-enter the bean. If the bean is in a transaction and a thread carrying a different transaction context attempts to enter the bean, the container will block the thread until the bean's transaction terminates. If the bean is not in a transaction when a thread attempts to enter the bean, and the thread is carrying a transaction of its own, the container will suspend the thread's current transaction and allow the thread to enter. The thread's original transaction is restored once the thread leaves the method, but any transaction created by method is not terminated by the container.

For both stateless session beans and entity beans, the bean's method is not allowed to return with its transaction still active. The container should catch this and throw an exception.

Leaving a transaction active across method calls is stateful, and is not allowed for stateless session beans. For similar reasons, entity beans are also not

 Copyright © 1999 Nova Laboratories - Reproduction Prohibited

allowed to maintain an open transaction state across method calls when the bean has declared the TX_BEAN_MANAGED transaction control.

Session Synchronization Interface

A session bean, stateless or stateful, is allowed to access a database, and can participate in transactions. To assist the bean in performing its work in a transaction, the bean developer can have the bean implement the javax.ejb.SessionSynchronization interface.

This interface is automatically detected by the container, and the container will use the methods in the interface as callbacks to keep the bean informed about the state of the transaction.

Entity beans do not support this interface. Since entity beans are implicitly transaction-aware, the container uses different semantics for controlling an entity bean while it is within a transaction.

The SessionSynchronization interface is defined as follows:

```
public interface javax.ejb.SessionSynchronization {
  public void afterBegin()       throws RemoteException;
  public void beforeCompletion() throws RemoteException;
  public void afterCompletion(boolean yn) throws
                                          RemoteException;
}
```

A transaction does not actually belong to a particular bean instance. A thread of execution creates a transaction, either in the client or in the container, and carries its transaction with it when executes the code in the bean. If a thread that has a transaction context is about to enter a session bean, the container first invokes its afterBegin() method. The bean can record the fact that all business method calls are running within a transaction, and subsequently allow certain transactional operations to execute. It would similarly reject requests to perform transactional operations if its internal flag indicated that this thread was not running within a transaction.

The bean would continue to assume that its business method calls are part of a transaction until the afterCompletion() method is called. It would presumably clear its internal flag indicating that subsequent requests that require transactions should be rejected.

If a thread that carries a transaction context attempts to enter a bean that is already a part of another transaction, the container will block the entry until the previous transaction either commits or rolls back and the afterCompletion() method is called, allowing the bean to restore its state. The container is responsible for providing these behaviors.

The beforeCompletion() method will be called on all session beans that are a part of a particular transaction when the container realizes that it is about to attempt to commit the transaction. This gives the bean the opportunity to finish its transactional operations, such as flushing any remaining data to the database, before the transaction is committed. If the container realizes it is about to perform a rollback instead, the beforeCompletion() method will not be called, since it is pointless to flush data in a transaction where the transaction is going to be rolled back, anyway.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

The afterCompletion() method is called after the transaction is either committed or rolled back to inform the bean about the final outcome of its transactional operations. It can use this information to update its internal state, but it should not use it to try to maintain any internal transactions it may be trying to maintain. While it is possible for a session bean to create, commit, and roll back its own transactions, it is generally discouraged. The SessionSynchronization interface does not provide the capability of functionally merging outer transactions with an inner transaction being maintained directly by the session bean.

A session bean that implements this interface implies that it is maintaining state across method calls. Specifically, the implication is that between the calls to afterBegin() and afterCompletion() the bean is within a transaction. Therefore, it is an error for a bean to implement the SessionSynchronization interface and be declared as stateless in its deployment descriptor. The vendor's installation tool should catch this and report an error.

Stateless session beans can participate in transactions, but they cannot implement this interface. The transaction could be TX_BEAN_MANAGED, or the container could start and commit the transaction upon entry to and return from the method. The container should not allow a thread with an existing transaction to enter the method, since the stateless bean's method would not know if the executing thread were running in the context of a transaction or not.

One way around this would be to have the container suspend an existing transaction, forcing the method to always assume that the thread is not running the method in a transactional context.

Stateful session beans can also participate in a transaction without implementing this interface. However, the deployment descriptor would have to be configured carefully so that the business methods always execute in the correct transaction state. The bean would not have the benefit of being informed of its transaction state through this interface.

Taking Part in Transactions

The EJBContext interface was illustrated in a previous section. In that interface were two methods:

```
public boolean getRollbackOnly();
public void    setRollbackOnly();
```

These methods can be used by any bean, not just beans that declare their transaction control as bean-managed. In fact, beans that manage their own transactions would not use these methods, since these methods are used to communicate transaction status to an outside transaction manager.

When a bean calls setRollbackOnly() it is requesting to the transaction manager that when it comes time to terminate the current transaction that the transaction be rolled back. It gives beans a vote in the outcome of the transactions in which they participate. These methods also exist in the UserTransaction interface, but since most beans will not have access to this interface, these methods must be provided to the bean directly in the EJBContext. Note that this method does not initiate a rollback, it simply sets a

 Copyright © 1999 Nova Laboratories - Reproduction Prohibited

flag that the transaction should be rolled back when the transaction is terminated.

Unlike JavaBeans™ property setter methods, this one does not take a `boolean` value as an argument. The method is designed this way deliberately so that a bean cannot revert the rollback request of another bean.

A bean may wish to use the `getRollbackOnly()` method to check the current status of the transaction. If another bean has already flagged the transaction for rollback, the calling bean could presumably decide not to perform intensive operations, such as database updates, that would be reverted once the transaction were terminated.

Client-Demarcated Transactions

While not a requirement for EJB vendors, a server vendor may decide to provide a class that can be used by the client to directly access the transaction manager. A client may wish to do this if it were necessary to invoke beans in two different servers in the same context. Of course, each bean's deployment descriptor would have to allow this behavior. The client could create a transaction and then invoke the business methods in two different beans on two different servers, passing the transaction context as part of the call. Once the calls were completed, the client would presumably terminate the transaction. The stubs and skeletons generated by the container vendor would have to support the implicit passing of a transaction context for this to work.

Here is a possible example:

```
Current current = new Current();
current.setServiceProvider(txMgrURL);
current.create();
current.begin();
remRef1.doSomeWork();
remRef2.doMoreWork();
current.commit();
```

Transaction Management of Database Operations

It is certainly desirable for a bean to be able to use JDBC™ to create connections to databases and to perform work on that connection. However, to fit into the EJB model of having the container manage the bean's transaction, the connection must not use the auto-commit feature, and the bean should not attempt a commit or rollback on the connection.

Presumably the container is the one that should determine if all transactional behavior executed within this transaction should be committed or rolled back.

A good question at this point is how the container could see and manage a database connection created internally by the bean's method. Although not stated explicitly in the Specification, enterprise beans should only use JDBC drivers that were intended for use with EJB. Upon creation of a database connection, these drivers will transparently register the connection with the executing thread's current transaction. Later when the container decides to terminate the transaction, the database connection will be automatically terminated with it.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited

In OTS terms, the database connection is a recoverable `Resource` that is implicitly managed by the transaction service with the help of the container.

While it is possible to use JDBC drivers that are not transaction-aware in this sense, the developer should be aware that any work done on this database connection is not a part of the bean's transaction, and the developer must be sure to terminate the database connection's transaction prior to returning from the method. Trying to use the `SessionSynchronization` interface to functionally merge the database connection's transaction with the bean's transaction is unreliable and should not be attempted.

Distributed Transaction Support

A distributed transaction would be needed in the following cases:

- a client using client-demarcated transactions that creates and transaction and calls a method in one or more beans on one or more servers
- a bean's method that invokes a method in another enterprise bean on another server

For this to work the vendor must generate stubs and skeletons for the EJBObject that will implicitly obtain the current transaction context and send it in the method call to the remote bean. The skeleton for the remote bean's EJBObject must apply this transaction context when delegating the business method call to the bean.

Copyright © 1999 Nova Laboratories - Reproduction Prohibited