Enterprise JavaBeans

1. Component software and distributed objects

Distributed objects can live anywhere on a network. They can be accessed via method invocations by clients, who need not be aware of the language and compiler used to create them, or on which machine or operating system the objects are executing. To achieve this transparency, distributed objects use services defined by a component model. Component models such as CORBA, EJB, and Microsoft's Distributed Component Object Model, differ in the level of functionality and quality they provide. Some services that are needed to achieve object transparency are: security, licensing, versioning, life-cycle management, support for open tool palettes, event notification, configuration and property management, scripting, meta-data and introspection, transaction control and locking, persistence, ease of use, and self-installation.

2. JavaBeans and Enterprise JavaBeans.

The JavaBeans component model specifies how components expose their properties, methods, and events. Although beans can use the Java Development Kit (JDK) as a component framework, a bean does not derive from some universal base class that gives it bean-like properties. Almost anything written in Java can be made into a bean. In fact, according to the JavaBeans specification, any Java class is a bean. This means that the JavaBeans specification defines no constraints. Introspection is what differentiates beans from ordinary Java classes. As long as a bean follows the defined conventions, a tool can look inside and discover its properties and behavior. The JavaBeans naming Conventions, also known as JavaBeans design patterns, include Conventions for naming simple properties, Boolean properties, indexed properties, multicast events, unicast events, and public methods. In addition, a developer can define a BeanInfo class to provide (via descriptors) all or parts of the bean's introspection information. This is optional, as the BeanInfo class can be generated for any bean that follows the naming conventions by using the Introspector class. The JavaBeans component model packaged with the JDK supports security (using the Java Security Manager), versioning, life-cycle management, support for open tool palettes, event notification, configuration and property management, scripting, meta-data and introspection, persistence (using serialization), ease of use (via the BeanBox, a simple test container), and is self-installing (via Java archive files).

In contrast, Enterprise JavaBeans or EJBs are remotely executable components or business objects deployed on the server. They have a protocol that allows them to be accessed remotely and this protocol also allows them to be installed or deployed on a particular server. They have a set of mechanisms that allow them to delegate major qualities of service, security, transactional behavior, concurrancy (the ability to be accessed by more than one client at a time), and persistence (how their state can be saved) to the container in which they are placed on the EJB server. They get their behavior from being installed in a container. Containers provide the different qualities of service so selecting the right EJB server is critical.

The EJB standard defines a contract for how server-side components interact with their container, in which the container acts as a framework that expects application-specific beans to behave according to a given set of rules. To satisfy some of these rules, an EJB component must derive from an appropriate base class. Naming conventions are not defined as explicitly in the EJB specification and, although introspection can be used at the time an EJB component is deployed, the preferred way of Communicating deployment information is via a deployment descriptor. A deployment descriptor is analogous to a BeanInfo class and is used by a deployment agent. EJB defines interfaces and functional behavior for security, life-cycle management, configuration and property management, meta-data and introspection, transaction control and locking, and persistence.

The major benefit is that the bean developer can stipulate, when the bean is built, what kind of behavior is needed, but not how it's done. Development is in two parts. One person develops the bean. He verifies that it works with the build tools and includes a deployment descriptor which identifies the kinds of quality of service behaviors needed. In the next step, another person can take that bean and use a deployment tool that reads the EJB deployment descriptor and installs the bean into a container on an enterprise Java server. In this second step, the deployment tool takes some action and this may mean generating codes like state saving codes, putting in transactional hooks, or doing security checks. All this action is generated by the deployment tool. The bean developer and deployer can be different people.

Any platform independent JavaBean can be adapted, through the use of a deployment tool, into a platform specific EJB that has the correct qualities of services available to meet the specific requirements of existing business systems and applications. This is why an EJB server is so important to integrate systems, networks and architectures.

Key features of the EJB technology are:

- EJB components are server-side components written entirely in the Java programming language
- EJB components contain business logic only - no System-level programming
- System-level services (i.e. "plumbing") such as transactions, security, Life-cycle, threading, persistence, etc. are automatically managed for the EJB component by the EJB server
- EJB architecture is inherently transactional, distributed, portable, multi-tier, scalable and secure
- Components are declaratively customized at deployment time (Can customize: transactional behavior, security features, life-cycle, state management, persistence, etc.)
- EJB components are fully portable across any EJB server and any OS
- EJB architecture is wire-protocol neutral - Any protocol can be utilized: IIOP, JRMP, HTTP, DCOM, etc.

Final Release of version 1.1 of the EJB specification has been made available in December 1999

3. EJB Base Structure

3.1 Services provided by a container for an Enterprise JavaBean component

The EJB server provides an environment that supports the execution of applications developed using Enterprise JavaBeans technology. It manages and coordinates the allocation of resources to the applications.

An EJB container constitutes a run-time ennvironment for multiple EJB´s. Some people refer to the container as a Framework or as an Object Transaction Monitor (OTM).

Enterprise beans are deployed in an EJB container within an EJB server. A container provides Enterprise Java Beans components with services of several types. First, it provides services for lifecycle management and instance pooling, including creation, activation, passivation, and destruction. Second it intercedes between client calls on the remote interface and the corresponding methods in a bean to enforce transaction and security constraints. It can provide notification at the start and end of each transaction involving a bean instance. Finally, it enforces policies and restrictions on bean instances, such as reentrance rules, security policies, and others.

The EJB container acts as a liaison between the client and the enterprise bean. At deployment time, the container automatically generates an EJB Home interface to represent the enterprise bean class and an EJB Object interface for each enterprise bean instance. The EJB Home interface identifies the enterprise bean class and is used to create, find, and remove enterprise bean instances. The EJB Object interface provides access to the business methods within the bean. All client requests directed at the EJB Home or EJB Object interfaces are intercepted by the EJB container to insert lifecycle, transaction, state, security, and persistence rules on all operations.

An EJB container manages the enterprise beans that are deployed within it. Client applications do not directly interact with an enterprise bean. Instead, the client application interacts with the enterprise bean through two wrapper interfaces that are generated by the container: the EJB Home interface and the EJB Object interface. As the client invokes operations using the wrapper interfaces, the container intercepts each method call and inserts the management services.

Figure 1 shows a representation of an EJB container. The EJB server must provide one or more EJB containers, which provide homes for the enterprise beans. An EJB container manages the enterprise beans contained within it. For each enterprise bean, the container is responsible for registering the object, providing a remote interface for the object, creating and destroying object instances, checking security for the object, managing the active state for the object, and coordinating distributed transactions. Optionally, the container can also manage all persistent data within the object.

Any number of EJB classes can be installed in a single EJB container. A particular class of enterprise bean is assigned to one and only one EJB container, but a container may not necessarily represent a physical location. The physical manifestation of an EJB container is not defined in the Enterprise JavaBeans specification. An EJB container could be implemented as a physical entity, such as a multithreaded process within an EJB server. It also could be implemented as a logical entity that can be replicated and distributed across any number of systems and processes.

An enterprise bean can be deployed in any EJB server, even though different servers implement their services in different ways. The EJB model ensures portability across different EJB servers using a set of standard contracts between the EJB container and the enterprise bean. Each enterprise bean is required to implement a specific set of interfaces that allows the EJB container to manage and control the object. The EJB container is required to invoke these interfaces at particular stages of execution.

There are two types of enterprise beans -- session beans and entity beans -- representing different types of business logic abstractions.

Session beans represent behaviors associated with client sessions -- they're generally implemented to perform a sequence of tasks within the context of a transaction. A session bean is a logical extension of the client program, running processes on the client's behalf remotely on the server.

Entity beans represent specific data or collections of data, such as a row in a relational database. Entity bean methods provide operations for acting on the data represented by the bean. An entity bean is persistent; it survives as long as its data remains in the database.

Enterprise JavaBeans technology supports both transient and persistent objects. A transient object is called a session bean, and a persistent object is called an entity bean.

3.2 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 can be transactional, but (normally) they are not recoverable following a system crash. Session beans can be stateless, or they can maintain conversational state across methods and transactions. The container manages the conversational state of a session bean if it needs to be evicted from memory. A session bean must manage its own persistent data.

A session bean encapsulates one or more business tasks and nonpermanent data associated with a particular client. Unlike the data in an entity bean, the data in a session bean is not stored in a permanent data source and no harm is caused if this data is lost. Nevertheless, a session bean can update data in an underlying database, usually by accessing an entity bean. For this reason, a session bean can be transaction aware. When created, instances of a session bean are identical, though some session beans can store semipermanent data that makes them unique at certain points in their life cycle. A session bean is always associated with a single client.

For example, the task associated with transferring funds between two bank accounts can be encapsulated in a session bean. Such a transfer enterprise bean might find two instances of an account enterprise bean (by using the account IDs), and then subtract a specified amount from one account and add the same amount to the other account.

The session bean developer defines the home and remote interfaces that represent the client view of the bean. Developers also create a class that implements both the SessionBean and SessionSynchronization interfaces, as well as methods corresponding to those in the bean's home and remote interfaces.

The tools for a container generate additional classes for a session bean at deployment time. These tools get information from the Enterprise JavaBeans architecture at deployment time by introspecting its classes and interfaces. They use this information to dynamically generate two classes, implementing the home and remote interfaces of the bean. These classes enable the container to intercede in all client calls on the session bean. The container also generates a serializable Handle class, providing a way to identify a session bean instance within a specific life cycle. These classes can be implemented to mix in container-specific code for performing customized operations and functionality.

In addition to these custom classes, each container provides a class to provide meta data to the client and implements the SessionContext interface to provide access to information about the environment in which a bean is invoked.

There is s distinction between a stateless and stateful session beans

Stateless beans are beans that don't maintain state across method calls. They're generally intended to perform individual operations atomically. They're also amorphous, in that any instance of a stateless bean can be used by any client at any time, at the container's discretion. They are the lightest weight and easiest to manage of the various Enterprise JavaBeans component configurations.

Stateful session beans maintain state within and between transactions. Each stateful session bean is associated with a specific client. Containers can automatically save and retrieve a bean's state in the process of managing instance pools of stateful session beans.

Stateful session beans maintain data consistency across transaction updates by updating their fields each time a transaction is committed. To keep informed of changes in transaction status, a stateful session bean implements the SessionSynchronization interface. The container then calls methods of this interface as it initiates and completes transactions involving the bean.

Session beans aren't designed to be persistent, whether stateful or stateless. The data maintained by a stateful session bean is intended to be transitional, solely for the purposes of a particular session with a particular client. A stateful session bean instance typically can't survive system failures and other destructuve events. While a session bean has a container-provided identity (called its handle), that identity passes when the session bean is removed by the client at the end of a session. If a client needs to revive a stateful session bean that has disappeared, it must provide its own means to reconstruct the bean's state.

3.3 Entity Beans

An entity bean represents data in a database and it provides shared access to multiple users. Entity beans are transactional and long-lived, and they survive crashes of the EJB server. An entity bean has associated methods to manipulate that data. In most cases, an entity bean must be accessed in some transactional manner. Instances of an entity bean are unique, and they can be accessed by multiple users.A primary key identifies each instance of an entity bean. Entity beans can be created either by inserting data directly into the database or by creating an object (using an object factory Create method). Entity beans are transactional, and they are recoverable following a system crash.

For example, the information about a bank account can be encapsulated in an entity bean instance. An account enterprise bean might contain an account ID, an account type (checking or savings), and a balance.

The entity bean developer defines the home and remote interfaces that represent the client view of the bean. Developers also create a class that implements the EntityBean interface, as well as methods corresponding to those in the bean's home and remote interfaces.

In addition to defining create methods in the EJBHome interface, the entity bean developer must also implement finder methods.

A finder method provides a way to access an entity bean by its contents. Finder methods are designed to be introspected and displayed by development and deployment tools. This enables a user to graphically manipulate entity beans in the process of developing applications.

The principal finder method that must be implemented by all entity beans is *findByPrimaryKey*. In addition to this method, the developer must also implement a PrimaryKey class to provide each entity bean with a unique, serializable identity.

As with session beans, the tools for a container generate additional classes for an entity bean at deployment time to implement the home and remote interfaces. These classes enable the container to intercede in all client calls on the same entity bean. The container also generates the serializable Handle class, providing a way to identify the entity bean within a specific life cycle. These classes can be implemented to mix in container-specific code for performing customized operations and functionality. In addition to these custom classes, each container provides a class to provide meta data to the client. Finally, where specified by a particular bean, a container manages persistence of selected fields of the entity bean.

An entity bean can implement its persistence directly, using bean-managed persistence, or by relying on its container, using container-managed persistence

In container-managed persistence, entity bean data is automatically maintained by the container using a mechanism of its choosing. For example, a container implemented on top of an RDBMS may manage persistence by storing each bean's data as a row in a table. Or, the container may use Java programming language serialization for persistence. When a bean chooses to have its persistence container managed, it specifies which of its fields are to be retained.

In bean-managed persistence, the bean is entirely responsible for storing and retrieving its instance data. The EntityBean interface provides methods for the container to notify an instance when it needs to store or retrieve its data.

An entity bean can be created in two ways: by direct action of the client in which a create method is called on the bean's home interface, or by some other action that adds data to the database that the bean type represents. In fact, in an environment with legacy data, entity objects may "exist" before an Enterprise JavaBean is even deployed.

A client can get a reference to an existing entity bean in several ways:

- receiving the bean as a parameter in a method call
- looking the bean up through a finder method of the home interface

obtaining the bean as a handle, a runtime specific identifier generated for a bean automatically by the container

In Release 1.0 of the Enterprise JavaBeans specification, support for session beans is required, but support for entity beans and container-managed persistence is optional. Mandatory support for these features will be required in a future version of the specification.

4. Enterprise JavaBeans component interfaces

4.1 EJB Home

The client view of an Enterprise JavaBeans component is provided through two interfaces -- the home interface and the remote interface. These interfaces are provided by classes constructed by the container when a bean is deployed, based on information provided by the bean. The home interface provides methods for creating a bean instance, while the remote interface provides the business logic methods for the component. By implementing these interfaces, the container can intercede in client operations on a bean, and offers the client a simplified view of the component.

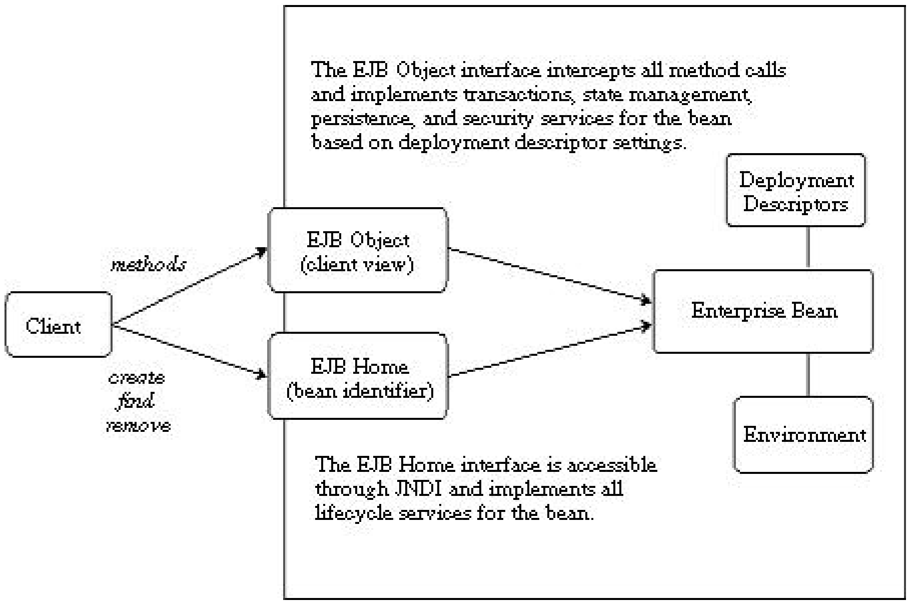
The EJB Object interface intercepts all method calls and implements transactions, state management, persistence, and security services for the bean based on deployment descriptor settings.
Deployment Descriptors
methods
EJB Object (client view)
Client
Enterprise Bean
EJB Home (bean identifier)
create find remove
Environment
The EJB Home interface is accessible through JNDI and implements all lifecycle services for the bean.

Figure 1.

To the client, there appears to be direct interaction with an Enterprise Java Bean through the home and remote interfaces. However, Enterprise JavaBeans architecture is designed to enable clients and components to exist in different runtimes on different systems on a network. Therefore the client does not interact with an Enterprise JavaBean directly. The container intercedes between client and component, completely concealing both the bean instance and its own actions from the clients.

The EJB Home interface provides access to the bean's lifecycle services. Clients can use the Home interface to create or destroy bean instances. For entity beans, the Home interface also provides one or more finder methods that allow a client to find an existing bean instance and retrieve it from its persistent data store.

For each class installed in a container, the container automatically registers the EJB Home interface in a directory using the Java Naming and Directory Interface (JNDI) API. Using JNDI, any client can locate the EJB Home interface to create a new bean instance or to find an existing entity bean instance. When a client creates or finds a bean, the container returns an EJB Object interface.

A client accesses an Enterprise JavaBean by looking up the class implementing its home interface by name through JNDI. It then uses methods of the home interface to acquire access to an instance of the class implementing the remote interface.

```
Context initialContext = new InitialContext() AccountHome accountHome =
javax.rmi.PortableRemoteObject.narrow(
initialContext.lookup("applications/bank/accounts"),
AccountHome.class);
```

In addition to generating classes to implement both the home and remote interface, the container is responsible for binding the home class to a JNDI name. This is generally handled automatically by the container tools at deployment time, without any user intervention required.

4.2 EJB Object

The (remote) EJB Object interface provides access to the business methods within the enterprise bean. An EJB Object represents a client view of the enterprise bean. The EJB Object exposes all of the application-related interfaces for the object, but not the interfaces that allow the EJB container to manage and control the object. The EJB Object wrapper allows the EJB container to intercept all operations made on the enterprise bean. Each time a client invokes a method on the EJB Object, the request goes through the EJB container before being delegated to the enterprise bean. The EJB container implements state management, transaction control, security, and persistence services transparently to both the client and the enterprise bean.

An Enterprise JavaBean's remote interface is written as an RMI remote interface. However, the managed object that implements it is CORBA IDL-based, so an RMI-to-IDL translation is required, which implies support for RMI/IIOP.

4.3 Declarative Attributes

The rules associated with the enterprise bean governing lifecycle, transactions, security, and persistence are defined in an associated Deployment Descriptor object. These rules are defined declaratively at deployment time rather than programmatically at development time. At runtime, the EJB container automatically performs the services according to the values specified in the deployment descriptor object associated with the enterprise bean.

4.4 Context Object

For each active enterprise bean instance, the EJB container generates an instance context object to maintain information about the management rules and the current state of the instance. A session bean uses a SessionContext object, and an entity bean uses an EntityContext object. The context object is used by both the enterprise bean and the EJB container to coordinate transactions, security, persistence, and other system services. Also associated with each enterprise bean is a properties table called the Environment object. The Environment object contains the customized property values set during the application assembly process or the enterprise bean deployment process.