10. ENTERPRISE JAVABEANS (EJBs)

Introduction

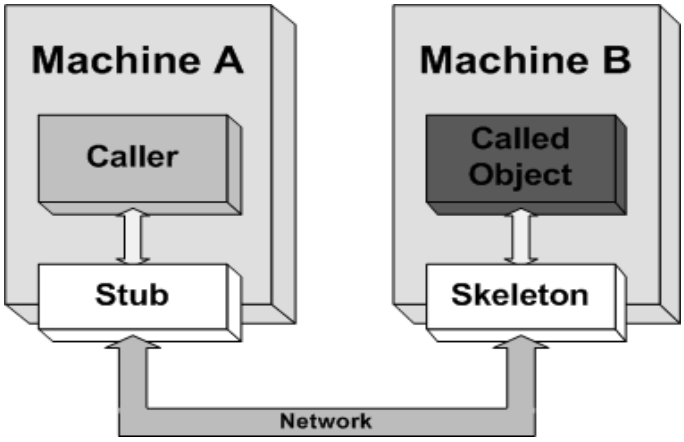
Distributed Computing

- It is peer to peer.
- It is networked computers providing single computer view to the user.
- It is ability to distribute the computing work load between clients and shared servers.
- It has more total computing power than a mainframe.
- Example:
  - Network of branch office computers
  - Network of work stations

Distributed Programming

In a distributed framework:

- Client makes a call to interface of a business object (i.e. **stub**).
- The stub then communicates this request to a **tie (also called skeleton)**.
- The tie calls the method on the real business object.
- A result is returned to the stub and the client.



Stub and Skeleton

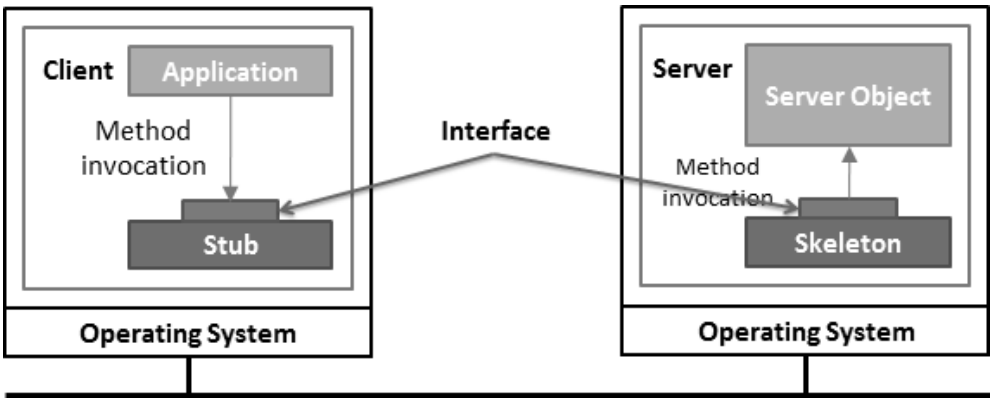
Stub:

- It is located at client side.
- It is gateway for client side objects and all outgoing requests to server side are routed through it.
- When the caller invokes method on the stub object, it does the following tasks:
  - It initiates a connection with remote Virtual Machine (JVM),
  - It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),

- It waits for the result
- It reads (unmarshals) the return value or exception, and
- It finally, returns the value to the caller.

Skeleton:

- It is located at server side.
- It acts as gateway for server side objects and all incoming client requests are routed through it.
- It does the following tasks:
  - It reads the parameter for the remote method
  - It invokes the method on the actual remote object, and
  - It writes and transmits (marshals) the result to the caller.



Distributed Objects Model

A distributed object based system is a collection of objects that isolates the requesters of services (clients) from the providers of services (servers) by a well-defined encapsulating interface.

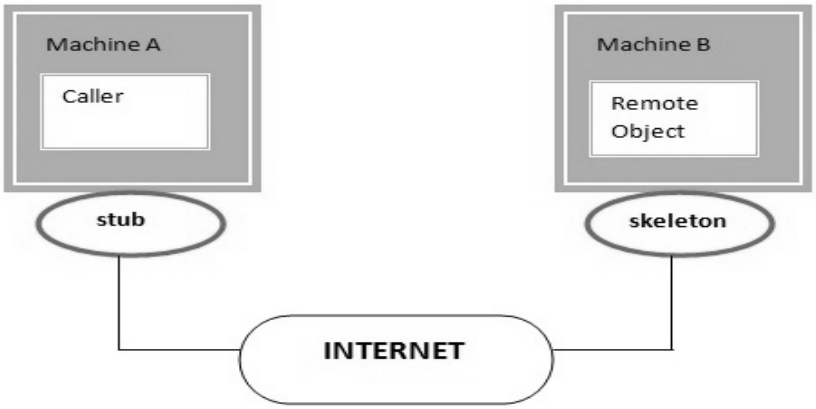
In the distributed object-based model, a client sends a message to an object, which in turn interprets the message to decide what service to perform. This service or method selection could be performed by either the object or a broker. The Java *Remote Method Invocation (RMI)* and the *Common Object Request Broker Architecture (CORBA)* are examples of this model.

RMI (Remote Method Invocation)

The RMI (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects stub and skeleton. RMI uses stub and skeleton object for communication with the remote object. A **remote object** is an object whose method can be invoked from another JVM.

Developing Distributed applications in RMI is simple than developing with sockets since there is no need to design a protocol which is an error-prone task. In RMI, the developer has the illusion of calling a local method from a local class file, when in fact the arguments are shipped to the remote target and interpreted and the results are sent back to the caller.



Steps to write the RMI program

The given are the 6 steps to write the RMI program.

- Create the remote interface
- Provide the implementation of the remote interface
- Compile the implementation class and create the stub and skeleton objects using the rmic tool
- Start the registry service by rmiregistry tool
- Create and start the remote application
- Create and start the client application

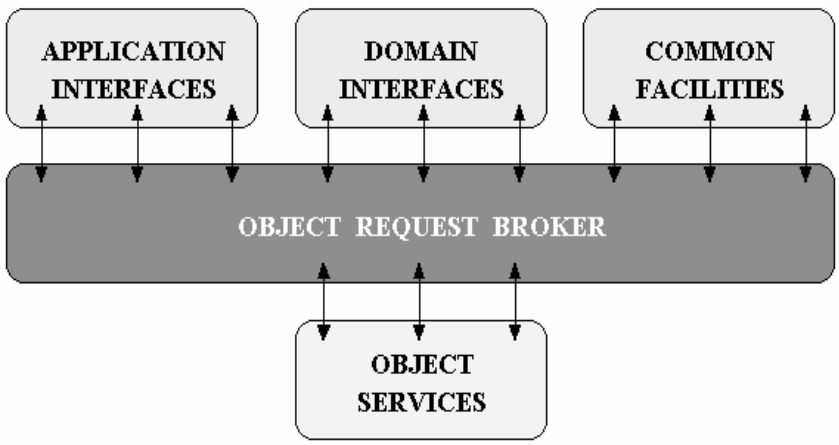
RMI provides a means of communicating between Java applications using normal method calls and offers the capability for the applications to run on separate computers-located perhaps as far apart as on opposite sides of the world. RMI application enables the client-server communications over the net.

CORBA (Common Object Request Broker Architecture)

- The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) designed to facilitate the communication of systems that are deployed on diverse platforms.
- CORBA enables collaboration between systems on different operating systems, programming languages, and computing hardware.
- CORBA has many of the same design goals as object-oriented programming: encapsulation and reuse. CORBA uses an object-oriented model although the systems that utilize CORBA do not have to be object-oriented. CORBA is an example of the distributed object paradigm.

ORB is an acronym for *Object Request Broker*, which is an object-oriented version of an older technology called Remote Procedure Call (RPC). An ORB or RPC is a mechanism for invoking operations on an object (or calling a procedure) in a different ("remote") process that may be running on the same or a different, computer. At a programming level, these "remote" calls look similar to "local" calls. The Object Request Broker (ORB) is the "middleware" which performs the communication. Objects may reside on the same computer, or different computers connected by a network.

The following figure illustrates the primary components in the ***OMG Reference Model architecture***.



- *Object Services* -- These are domain-independent interfaces that are used by many distributed object programs. For example, a service providing for the discovery of other available services is almost always necessary regardless of the application domain. Two examples of Object Services that fulfill this role are:
    - *Naming Service* -- which allows clients to find objects based on names;
    - *Trading Service* -- which allows clients to find objects based on their properties.

  There are also Object Service specifications for lifecycle management, security, transactions, and event notification, as well as many others.
- *Common Facilities* -- Like Object Service interfaces, these interfaces are also horizontally-oriented, but unlike Object Services they are oriented towards end-user applications.
- *Domain Interfaces* -- These interfaces fill roles similar to Object Services and Common Facilities but are oriented towards specific application domains.
- *Application Interfaces* - These are interfaces developed specifically for a given application. Because they are application-specific, and because the OMG does not develop applications (only specifications), these interfaces are not standardized.

One of CORBA's strong points is that it is distributed middleware. In particular, it allows applications to talk to each other even if the applications are:

- On different computers, for example, across a network.
- On different operating systems. CORBA products are available for many operating systems, including Windows, UNIX, IBM mainframes and embedded systems.
- On different CPU types, for example, Intel, SPARC, PowerPC, big-endian or little-endian, and different word sizes, for example, 32-bit and 64-bit CPUs.
- Implemented with different programming languages, such as, C, C++, Java, Smalltalk, Ada, COBOL, PL/I, LISP, Python and IDLScript.

Enterprise JavaBeans (EJBs)

EJB (Enterprise Java Bean) is used to develop scalable, robust and secured enterprise applications in java. Unlike RMI, middleware services such as security, transaction management etc. are provided by EJB Container to all EJB applications.

What is EJB?

It is a specification provided by Sun Microsystems to develop secured, robust and scalable distributed applications. To run EJB application, you need an application server (EJB Container) such as Jboss, Glassfish, Weblogic, Websphere etc. It performs:

- life cycle management,
- security,
- transaction management, and
- object pooling.

EJB application is deployed on the server, so it is called server side component also. EJB is like COM (Component Object Model) provided by Microsoft. But, it is different from Java Bean, RMI and Web Services.

When use Enterprise Java Bean?

- *Application needs Remote Access*. In other words, it is distributed.
- *Application needs to be scalable*. EJB applications supports load balancing, clustering and fail-over.
- *Application needs encapsulated business logic*. EJB application is separated from presentation and persistent layer.

Types of Enterprise Java Bean

There are 3 types of enterprise bean in java.

- *Session Bean*: Session bean contains business logic that can be invoked by local, remote or web service client.
- *Message Driven Bean*: Like Session Bean, it contains the business logic but it is invoked by passing message.

- *Entity Bean*: It encapsulates the state that can be persisted in the database. It is deprecated. Now, it is replaced with JPA (Java Persistent API).

Difference between RMI and EJB

RMI	EJB
In RMI, middleware services such as security, transaction management, object pooling etc. need to be done by the java programmer.	In EJB, middleware services are provided by EJB Container automatically.
RMI is not a server-side component. It is not required to be deployed on the server.	EJB is a server-side component, it is required to be deployed on the server.
RMI is built on the top of socket programming.	EJB technology is built on the top of RMI.

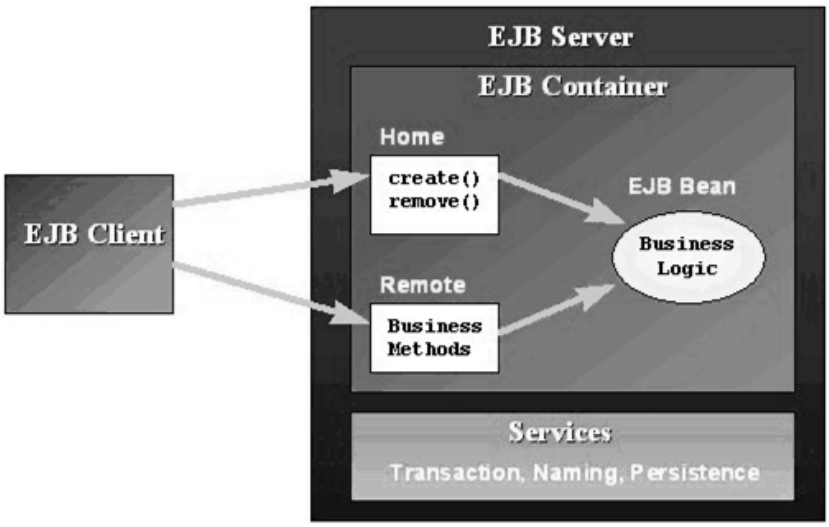
Disadvantages of EJB

- Requires application server
- Requires only java client. For other language client, you need to go for web service.
- Complex to understand and develop ejb applications.

Viewing the EJB Framework

When you create a distributed application with Enterprise JavaBeans technology, you code the business logic in reusable components called enterprise beans. Enterprise beans reside in an EJB container that runs on an EJB server.

The EJB framework consists of components and services. In the diagram below you can see the location of those components. The EJB server is the highest level component, including all the others, and providing the EJB container the access to the services. The home interface is returning a reference on the enterprise bean that is invoked and manipulated by the remote interface.



EJB Client

The EJB client is connects to the EJB server. It uses Java Naming and Directory Interface (JNDI) to locate the home interface object to create an EJB class instance. It then uses the remote interface of the EJB class instance to invoke methods on the server object. The EJB client and the EJB server communicate using Internet Inter-ORB Protocol (IIOP) or another protocol, such as Remote Method Invocation (RMI).

Enterprise Bean

The enterprise bean is a Java class or collection of classes, provided as a Java ARchive (JAR) file. The enterprise bean component takes advantage of the Enterprise JavaBean architecture extending classes aligned with the EJB specifications develops it.

EJB Container

The EJB container acts as the enterprise bean holder that manages the enterprise bean's classes and instances. EJB classes are deployed to the EJB container that manages the startup and shutdown of the EJB class instances. The EJB container provides system-level services to the EJB classes through the standard EJB APIs. The EJB server, which provides the basic system services, manages the EJB containers.

- **Home interface**
  Home interface defines the methods for locating, creating, and removing instances of the EJB classes. You must create and implement the required factory methods so that the EJB container can create instances of the EJB classes when a client establishes a connection to the server. The factory is a mandatory component of the distributed computing that returns a reference on an object.
- **Remote interface**
  EJB clients do not invoke methods on the EJB class directly, but instead go through the remote interface. Therefore, an EJB client can only define the business methods that a client may call through the remote interface. The business methods are implemented in the enterprise bean code.

EJB server

The EJB server manages EJB containers. It provides the underlying environment for an enterprise bean. It provides access to the basic system services required by the Enterprise JavaBeans architecture and any vendor specific features, such as optimized database access.

Deployment descriptor

The deployment descriptor is a file that contains all the information needed by the EJB container at runtime.

Services Provided by the EJB Container

The basic services provided by the EJB container are *transactions, naming, and persistence*.

- The ***transaction service*** allows multiple enterprise beans to participate in a transaction without transaction code implemented explicitly into the client's source.
- The ***naming service*** allows EJB clients to locate EJB servers by name.
- The ***persistence service*** allows the enterprise bean instances to be stored inside a database or in a file system.

Comparing JavaBeans and EJBs

Both JavaBeans and Enterprise JavaBeans are written in Java. However, they reside in different types of containers and are not interchangeable. There are other differences that are displayed in the following table:

JavaBeans...	Enterprise JavaBeans...
Contain a specification for creating and reusing components.	Contain a specification for a services framework so that you can deploy components.
Create the server framework.	Server framework is provided.
Contain EJB classes that must conform to the JavaBean specifications.	Contain EJB classes that must conform to the EJB specifications.

Session Beans

Session bean encapsulates business logic only; it can be invoked by local, remote and web-service client. It can be used for calculations, database access etc. The life cycle of session bean is maintained by the application server (EJB Container).

A session bean is the enterprise bean that directly interacts with the user and contains the business logic of the enterprise application. A session bean represents a single client accessing the enterprise application deployed on the server by invoking its method.

For example, whenever a client wants to perform any of these actions such as making a reservation or validating a credit card, a session bean should be used.

Types of Session Bean / Attributes of Beans: There are 3 types of session bean.

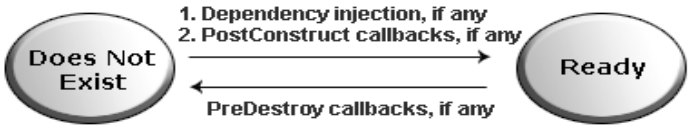
1) *Stateless Session Bean*: It doesn't maintain state of a client between multiple method calls.
2) *Stateful Session Bean*: It maintains state of a client across multiple requests.
3) *Singleton Session Bean*: One instance per application, it is shared between clients and supports concurrent access.

The session bean decides what data is to be modified. Typically, the session bean uses an entity bean to access or modify data. Entity bean implement business logic, business rules, algorithms, and work flows. Session beans are relatively short-lived components. The EJB container may destroy a session bean if its client times out.

Stateless Session Bean

- Stateless Session bean is a business object that represents business logic only. It doesn't have state (data). A stateless session bean does not maintain a conversational state with the client. So they do not persist data across method invocation and therefore there is no need to passivates the bean's instance and they are suitable for enterprise applications with more number of clients.
- The stateless bean objects are pooled by the EJB container to service the request on demand. It can be accessed by one client at a time. In case of concurrent access, EJB container routes each request to different instance.

Life Cycle of Stateless Session Beans



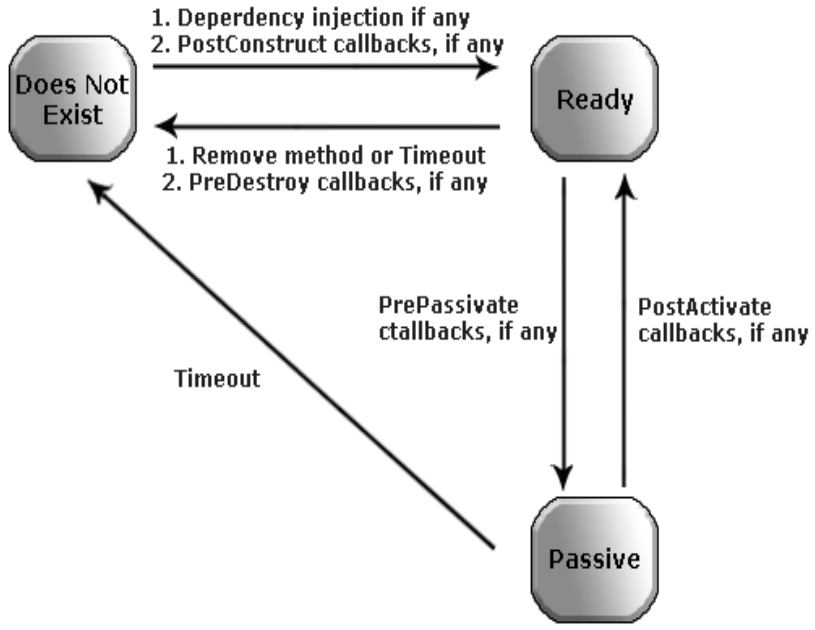
- Since the Stateless session bean does not passivates across method calls therefore a stateless session bean includes only two stages. Whether it ***does not exist*** or ***ready for method invocation***. A stateless session bean starts its life cycle when the client first obtains the reference of the session bean.
- EJB Container maintains the life cycle with the help of the following annotated methods
    - 1. **@PreConstruct** : if doesn't exist, this method will be called
    - 2. **@PreDestroy** : if client session ends, this will be called.

Stateful Session Beans

- Stateful Session bean is a business object that represents business logic like stateless session bean. But, it maintains state (data). In other words, conversational state between multiple method calls is maintained by the container in stateful session bean.
- These types of beans use the instance variables that allows the data persistent across method invocation because the instance variables allow persistence of data across method invocation.
- A stateful session bean retains its state across multiple method invocations made by the same client. The state of a client bean is retained for the duration of the

client-bean session. Once the client removes the bean or terminates, the session ends and the state disappears. Because the client interacts with its bean, this state is often called the conversational state.

Life Cycle of Stateful Session Beans



- A Stateful session bean starts its life cycle when the client first gets the reference of a stateful session bean.
- Before invoking the method annotated **@PostConstruct** the container performs any dependency injection after this the bean is ready.
- The container may deactivate a bean while in ready state (using LRU). In the passivate mechanism the bean moves from memory to secondary memory.
- The container invokes the annotated **@PrePassivate** method before passivating the bean.
- If a client invokes a business method on the passivated bean then the container invokes the annotated **@PostActivate** method to let come the bean in the ready state.
- At the end of the life cycle of the bean, the client calls the annotated **@Remove** method after this the container calls the annotated **@PreDestroy** method which results in the bean to be ready for the garbage collection.

Entity Beans

Entity beans are server-side components that are persistent and transactional. They are used to model persistent data objects. An entity bean can manage its own persistent state in a datastore and its own relationships, in which case it is a bean-managed persistence

(BMP) entity bean. It can let the EJB container manage its persistent state and relationships, in which case it is a container-managed persistence (CMP) entity bean. An entity bean represents a business object in a persistent storage mechanism. Some examples of business objects are

- customers,
- orders, and
- products.

Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table.

Features of Entity Bean

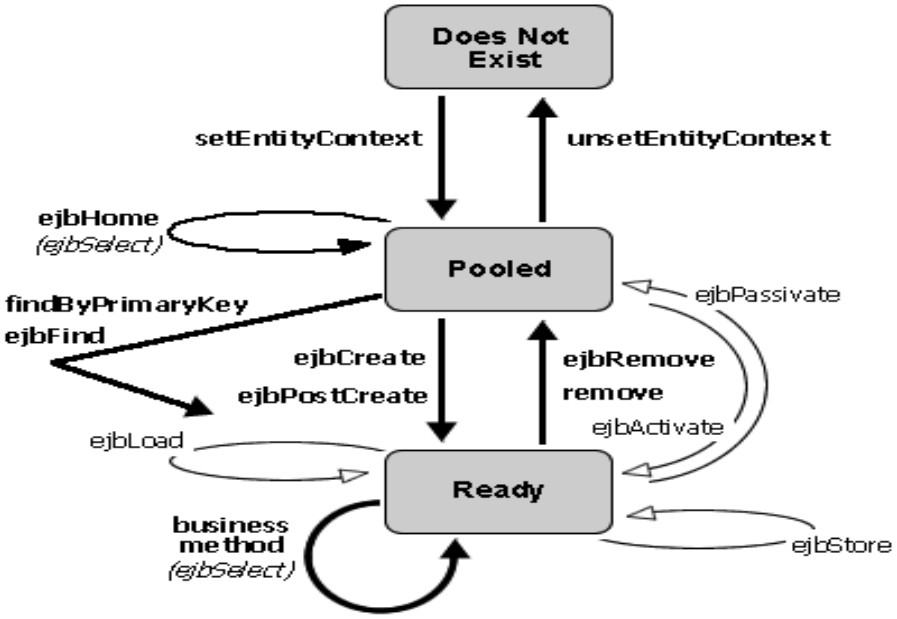
- *Persistence*
  - Because the state of an entity bean is saved in a storage mechanism, it is persistent. *Persistence* means that the entity bean's state exists beyond the lifetime of the application or the Application Server process.
- *Shared Access*
  - Entity beans can be shared by multiple clients. Because the clients might want to change the same data, it's important that entity beans work within transactions.
- *Primary Key*
  - Each entity bean has a unique object identifier.
- *Relationships*
  - Like a table in a relational database, an entity bean may be related to other entity beans. For example, in a college enrollment application, StudentBean and CourseBean would be related because students enroll in classes.

Life Cycle of Entity Bean

- When developing an entity bean we can take advantage of the bean's relationship with the container to execute logic and optimizations outside the context of the bean's core logic.
- As the container creates and pools an instance, assigns data to it, executes bean methods, and eventually removes the instance, the container provides opportunities for our code to execute.
- The following figure shows the life cycle of an entity bean. An entity bean has the following three states:
  - **Does not exist**. In this state, the bean instance simply does not exist.
  - **Pooled state** . When application server is first started, several bean instances are created and placed in the pool. A bean instance in the pooled

state is not tied to particular data, that is, it does not correspond to a record in a database table. Additional bean instances can be added to the pool as needed, and a maximum number of instances can be set.

– **Ready state**. A bean instance in the ready state is tied to particular data, that is, it represents an instance of an actual business object.



State transition in the life cycle

- Moving from the Does Not Exist to the Pooled State :
  - setEntityContext
- Pooled State: ejbSelect
- Moving from the Pooled to the Ready State
  - ejbCreate
  - ejbPostCreate
  - findByPrimaryKey
- Ready State
  - ejbLoad
  - ejbStore
- Moving from the Ready to the Pooled State
  - ejbRemove
- Activation and Passivation
  - ejbPassivate
  - ejbActivate
- Moving from the Pooled to the Does Not Exist State
  - unsetEntityContext

The following table provides an overview of the *differences between entity beans and session beans*:

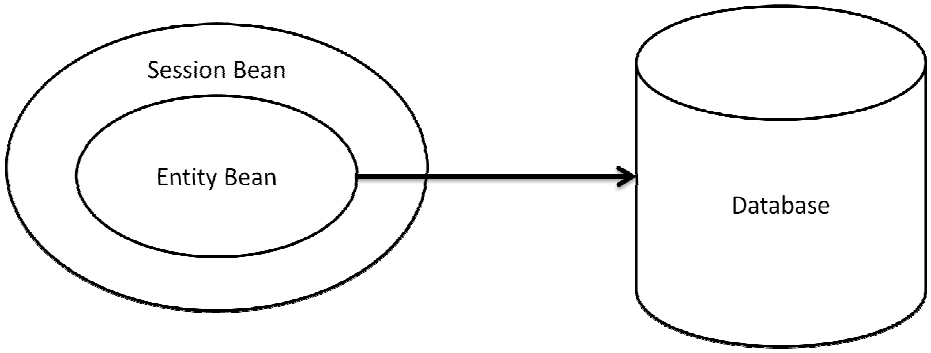
Entity Beans...	**Session Beans...**
Shared by multiple clients	Represents a single client inside the EJB server.
Represents a business object in a persistent storage mechanism such as a database. The persistence can be managed by either the entity bean itself, or by the EJB container. Bean-managed persistence requires you to write the data access code in the Bean	Not persistent. When the client terminates, its session bean appears to terminate and is no longer associated with the client.
Represents a business entity. Entity beans are server-side components where each instance of a bean maps to an underlying data object such as a row in the database.	Performs a task for a client. Session beans are server-side components where each instance of the bean maps to a client connection to the server.

Encapsulating Entity Beans with Session Beans

Encapsulating an entity bean inside a session bean is usually a good design choice for several reasons:

1. It will prevent a remote client from starting a transaction.
2. It will eliminate direct exposure of the entity beans to remote clients.

This makes entity beans to be more reusable.



Parts of a Bean

Some of the components used to create, access, describe and work alongside business logic contained in beans classes are:

- *Home Interface* EJBHome object that provides the life cycle operations (create, remove, find) for an EJB. EJBHome interface provides access to bean's life cycle services and can be utilized by a client to create or destroy bean instances.
- *EJBObject Interface* EJBObject interface provides access to the business methods within the EJB. An EJBObject represents a client view of the EJB. It allows the EJB container to intercept all operations made on the EJB.

- *Deployment Descriptor*
  The deployment descriptor is an XML file provided with each module and application that describes how the parts of a J2EE application should be deployed. It configures specific container options in your deployment tool of choice.
- *SessionContext and EntityContext Objects*
  For each active EJB 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, while an entity bean uses an EntityContext object. Both the EJB and the EJB container use the context objet to coordinate transactions, security, persistence and other system services.
- *Dependent and Fine-Grained Objects*
  There is a natural conflict between a large many in a one-to-many object relationship and a distributed framework. The EJB framework is not adapted to fine-grained objects, which may be associated with a large number of related objects. In order to ensure performance when passing fine-grained objects over a distributed framework, the local interface was created to allow intercontainer objects to be passed by reference instead of by value.

Container-Managed Persistence and Bean-Managed Persistence

Entity EJBs support two persistence options: *Bean Managed Persistence* (BMP) and *Container Managed Persistence* (CMP).

Bean-Managed Persistence (BMP) occurs when the entity object manages its own persistence. The enterprise bean developer must implement persistence operations (e.g., JDBC, JDO, or SQLJ) directly in the enterprise bean class methods.

With **BMP**, an entity bean contains its own database access code. This code includes methods that execute SQL commands against the database. For example, a bean might include methods that create new bean instances (SQL INSERT) as well as methods that retrieve existing bean instances from the database (SQL SELECT). BMP can enhance a bean's performance because it allows the database access code to be optimized for a specific database. Of course, developing the database access code can be very labor intensive and can involve complex object-relational mappings.

Container-Managed Persistence (CMP) occurs when the entity object delegates persistence services. With CMP, the EJB container transparently and implicitly manages the persistent state. The enterprise bean developer does not need to code any database access functions within the enterprise bean class methods.

With **CMP**, an entity bean contains no database access code. Instead, the EJB container generates all of the required database access code based upon a specification provided in

an XML file called the **abstract schema**. CMP enhances a bean's reusability by removing any code that might be database-specific from the bean. CMP also speeds development by freeing developers from having to create the database access code.

The following table provides a summary comparison of BMP and CMP:

	Efficiency	Development Effort	Reusability
BMP	Higher	Higher	Lower
CMP	Lower	Lower	Higher

JMS Tutorial

JMS (Java Message Service) is an API that provides the facility to *create, send and read* messages between two or more clients. It provides loosely coupled, reliable and asynchronous communication. JMS is also known as a messaging service.

Understanding Messaging

Messaging is a technique to communicate applications or software components. JMS is mainly used to send and receive message from one application to another.

Requirement of JMS

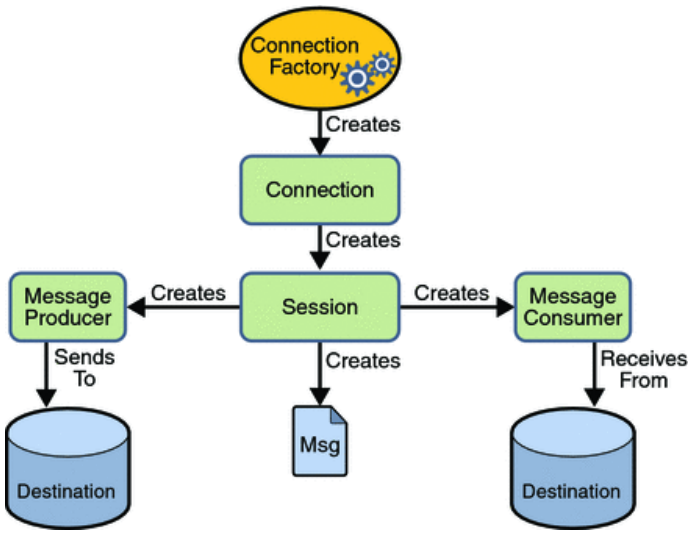
Generally, user sends message to application. But, if we want to send message from one application to another, we need to use JMS API.

Consider a scenario, one application A is running in INDIA and another application B is running in USA. To send message from A application to B, we need to use JMS.

Advantage of JMS

1) **Asynchronous:** To receive the message, client is not required to send request. Message will arrive automatically to the client.
2) **Reliable:** It provides assurance that message is delivered.

JMS Programming Model



The following are **JMS elements**:

JMS provider

An implementation of the JMS interface for a Message Oriented Middleware (MOM). Providers are implemented as either a Java JMS implementation or an adapter to a non-Java MOM.

JMS client

An application or process that produces and/or receives messages.

JMS producer/publisher

A JMS client that creates and sends messages.

JMS consumer/subscriber

A JMS client that receives messages.

JMS message

An object that contains the data being transferred between JMS clients.

JMS queue

A staging area that contains messages that have been sent and are waiting to be read (by only one consumer). Note that, contrary to what the name queue suggests, messages don't have to be received in the order in which they were sent. A JMS queue only guarantees that each message is processed only once.

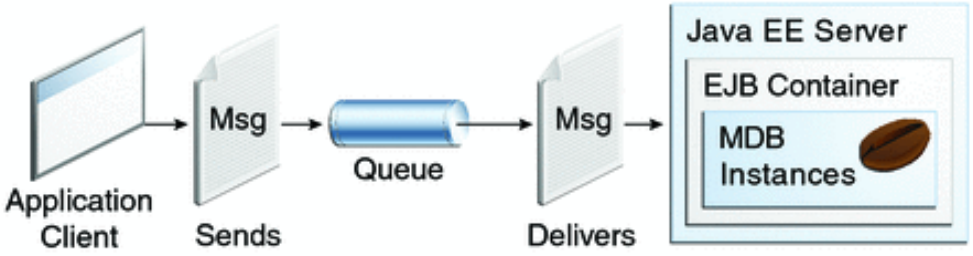
JMS topic

A distribution mechanism for publishing messages that are delivered to multiple subscribers.

Message Driven Bean (MDB)

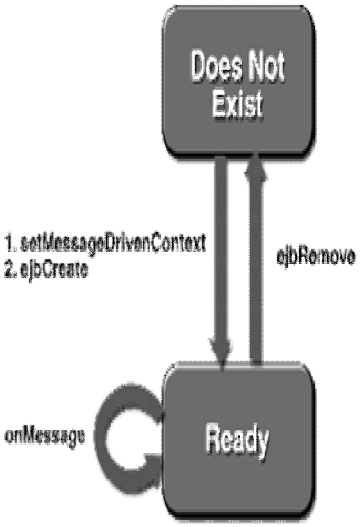
A message driven bean (MDB) is a bean that contains business logic. But, it is invoked by passing the message. So, it is like JMS Receiver. MDB asynchronously receives the message and processes it.

A message driven bean receives message from queue or topic, so you must have the knowledge of JMS API.A message driven bean is like stateless session bean that encapsulates the business logic and doesn't maintain state.



Message driven beans are the light weight components used for communication. In message driven beans the messaging service is in asynchronous mode because the user is not intended to get the instant result.

Life Cycle of a Message-Driven Bean



- The figure illustrates the stages in the life cycle of a message-driven bean.
- The EJB container usually creates a pool of message-driven bean instances. For each instance, the EJB container instantiates the bean and performs these tasks:
  - It calls the *setMessageDrivenContext* method to pass the context object to the instance.
  - It calls the instance's *ejbCreate* method.
  - At the end of the lifecycle, the container calls the method annotated *@PreDestroy*, if any. The bean's instance is then ready for garbage collection.

A MDB acts as a message consumer which receives messages from a JMS topic and performs business logic based on message contents. To receive JMS messages, MDBs implement the javax.jms.MessageListener interface, which defines a single onMessage() method.

```
public void onMessage(javax.jms.Message message) {
        //your code here
}
```

When a message is sent to a topic, the topic's Container ensures that the MDB corresponding to the topic does exist. If the MDB needs to be instantiated, the container will do this. The onMessage() methods initiates the business logic that processes the message. The EJB deployer is the person responsible for having MDBs assigned to a Topic at deployment time. The container provides the service of creating and removing MDB instances as they are needed.

Containers and MDBs

The EJB container or application server provides many services for MDBs so the developer can concentrate on implementing business logic. Some of the services that the container provides are:

- handles communication for JMS messages.
- checks its pool of available instances in order to see which MDB should be used.
- Enables the propagation of a security context by associating the role designated in the deployment descriptor to the appropriate thread of execution.
- Creates an association with a transactional context if one is required by the deployment descriptor.

- Passes the JMS message to the onMessage() method of the appropriate MDB instance.
- Re-allocates MDB resources to a pool of available instances.

Transactions and Transaction Management

A transaction is one or more tasks that execute as a single atomic operation or unit of work. If all tasks involved in a transaction do not proceed successfully then an inverse task or rollback procedure for all tasks is performed, setting all resources back to their original state. Transactions are characterized by the acronym ACID stands for (Atomic, Consistent, Isolated and Durable).

The EJB container provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization and transaction context propagation.

Since JDBC operates at the level of an individual database connection, it does not support transactions that span across multiple data sources. To compensate, Java Transaction API (JTA) provides access to the services offered by transaction manager.

Distributed Transactions

Although EJB framework can be used to implement non-transactional systems, the model was designed to support distributed transactions. EJB framework requires the use of a distributed transaction management system that supports two-phase commit protocols for flat transactions. An EJB may participate in client managed and bean managed transactions. The EJB architecture provides automatic support for distributed transactions in component based applications.

J2EE compliant containers support the following transaction attributes of EJBs:

- NotSupported: *bean runs outside the context of a transaction*
- Required: *method calls require a transaction context*
- Supports: *method calls use the current transaction context if one exists but don't create one if none exists.*
- RequiresNew: *containers create new transactions before each method call on the bean and commit transactions before returning.*
- Mandatory: *method calls require a transaction context.*
- Never: *method calls require that no transaction context be present.*

Multiple Transactions

A container can manage multiple transactions in 2 different ways: The container could instantiate multiple instances of the bean, allowing the transaction management of the DBMS to handle any transaction processing issues. Conversely, the container could

acquire an exclusive lock on the instance's state in the database, serializing access from multiple transactions to this instance.

Java Transaction Service (JTS)

The JTS specifies the implementation of a transaction manager supporting the JTA. JTS also implements the Java mapping of the OMG Object Transaction Service (OTS). JTS supports distributed transactions, which have the ability to span multiple databases on multiple systems coordinated by multiple transaction managers.

Java Transaction API (JTA)

EJB applications communicate with a transaction service using the Java Transaction API (JTA). JTA provides a programming interface to start transactions, to join existing transactions, to commit transactions and to roll back transactions.

Entity Bean Methods and Transaction Attributes

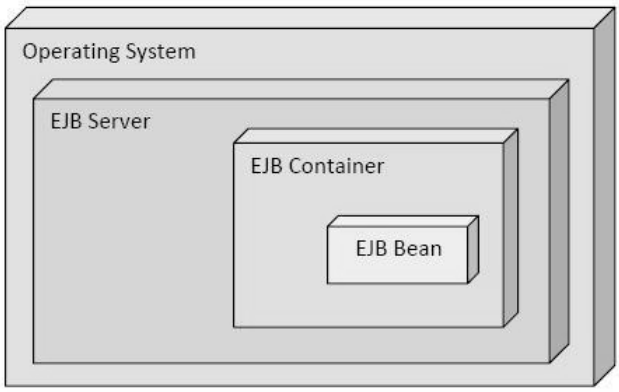
All developer-defined methods in the Remote interface as well as all methods defined in the Home interface (such as create, remove, and finder methods) require transaction attributes.

Session Bean Methods and Transaction Attributes

All developer-defined methods in the Remote interface require transaction attributes. Note that transaction attributes are not needed for the methods in the Home interface. Methods in the Remote interface run with the NotSupported attribute by default.

Roles in EJB

EJB specification gives a clear description of each of these roles and their associated responsibilities.



a) Enterprise Bean Provider

- It is the person or group that writes and packages EJBs.
- It might be a third-party vendor of EJB components, or it might be an information systems programmer who writes EJBs to implement their company's business logic.

- Because the container must provide transaction and network communication support, the enterprise bean provider does not need to be an expert at low-level system programming, networking or transaction processing.
- The end product is an ejb-jar file, which contains the Enterprise bean, its supporting classes, and information that describes how to deploy the bean.

b) Deployer

- The deployer takes an ejb-jar file and installs it into an EJB container.
- The deployer's task begins with the receipt of an ejb-jar file, and ends with the installation of the ejb-jar file in the container.
- This task is typically handled by the System Administrator in MIS world.

c) Application Assembler

- An application assembler builds applications using EJBs classes.
- An MIS programmer may purchase a prepackaged set of EJBs that implement an accounting system.
- Then the programmer might use tools to customize the beans.
- The application assembler might also need to provide a user-interface client program.
- This role is roughly analogous to that of a Systems Integrator in the MIS world.

d) EJB Server Provider

- An EJB Server Provider provides a server that can contain EJB containers.
- The EJB 1.0 specification does not describe the interface between the server and the container, so the EJB server provider and the EJB container provider will likely be one and the same for any given product.
- The diagram shows the relationships among the EJB server, the EJB container, and the EJB bean.
- An EJB server provides services to an EJB container and an EJB container provides services to an Enterprise Java Bean.
- The role of the EJB server provider is similar to that of a database system vendor.
- EJB Provider

e) EJB Container Provider

- An EJB container is the world in which an EJB bean lives.
- The container services requests from the EJB, forwards requests from the client to the EJB, and interacts with the EJB server.
- The container provider must provide transaction support, security and persistence to beans.
- It is also responsible for running the beans and for ensuring that the beans are protected from the outside world.

f) System Administrator

- System Administrators are responsible for the day-to-day operation of the EJB server.
- Their responsibilities include keeping security information up-to-date and monitoring the performance of the server.

Uses of Entity Beans, Stateful Beans, and Stateless Beans

- Entity beans are used to persist the enterprise data. The entity bean is a sharable enterprise data object that can be used by many users concurrently.
- The stateful session bean is used to manage the state of a users session on the server. An example of a stateful session bean would be a shopping cart.
- The stateless session bean can be used to implement the business logic and workflow of an application

Defining the Session Bean Class

The session bean class must be declared with the **`public`** attribute. This attribute enables the container to obtain access to the session bean.

Session Bean Interface

Session beans are held to the J2EE EJB specification that requires all session beans to implement javax.ejb.SessionBean interface. This requirement forces session beans to contain the following methods:

- ejbActivate()
- ejbPassivate()
- ejbRemove()
- setSessionContext(SessionContext)

Clients of a session bean may either be remote or local, depending on what interfaces are implemented.

Neither local nor remote clients access session beans directly. To gain access to session bean methods, they use a *component* interface to the session bean. Instances of a session bean's remote interface are called session EJBObjects, while instances of a session bean's local interface are called session EJBLocalObjects.

Both local and remote interfaces provide the following services to a client:

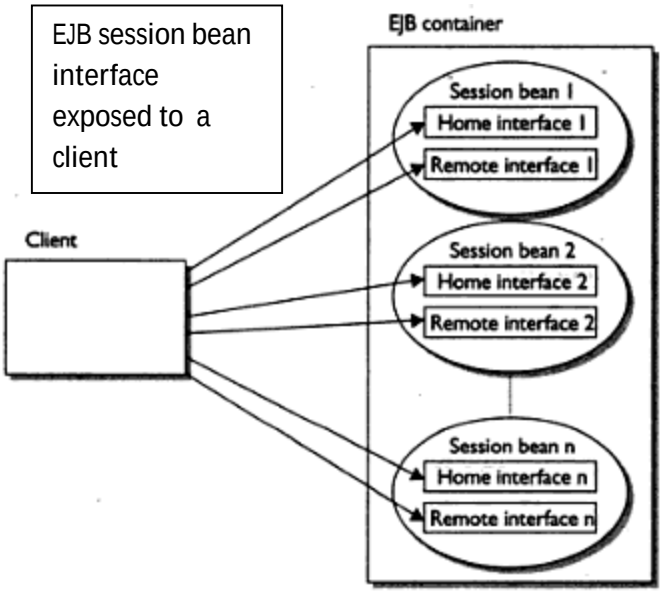
- Delegate business method invocations on a session bean instance
- Return the session object's home interface
- Test to determine whether a session object is identical to another session object
- Remove a session object

javax.ejb.SessionBean INTERFACE METHOD SUMMARY

Method	Description
`public void ejbActivate() throws EJBException, java.rmi.RemoteException`	Invoked by the container when the bean instance becomes activated after being in the "passive" state. Beans can use this method to re-access resources that it released before being *passivated* (made passive). See the `ejbPassivate` method.
`public void ejbPassivate() throws JBException, java.rmi.RemoteException`	Invoked by the container before the bean instance enters a "passive" state, at which point the bean may be serialized and persisted by the container (the container uses the Java Serialization protocol to externalize and save the instance's state to permanent storage). Beans should use this method to ensure that they're ready to become passive. When this method is invoked, the instance should release any resources that it can access again later when it becomes active (such resources should be re-accessed in the `ejbActivate` method).
`public void ejbRemove() throws EJBException, java.rmi.RemoteException`	Invoked by the container when the bean reaches the end of its life and should be removed. This typically happens when the client invokes a remove method or when the container chooses to terminate the session object because of an inactivity timeout (such as when the user doesn't interact with the client for a period of time).
`public void setSessionContext (SessionContext ctx) throws EJBException, java.rmi.RemoteException`	Invoked by the container after the bean instance has been created to set the session context object that will be associated with the bean over its lifetime (the `javax.ejb.SessionContext` interface that defines this session context object provides beans with access to their container's runtime environment). Beans can use this method to access their `SessionContext`, which can then be stored in an instance variable should the bean need to interact with the object during the course of execution.

When the application is deployed, the container or application server will use the interfaces defined by the enterprise bean provider and create EJBHome, EJBObject, stub, and tie classes:

- The *EJBHome* class is used to create instances of the session bean class and the *EJBObject* class.
- The *EJBObject* class provides access to the desired methods of the session bean.
- The stub classes act as proxies to the remote EJBObjects.
- The tie classes provide the call and dispatch mechanisms that bind the proxy to the EJBObject.



Methods in the session bean home interface