UNIT-IV

ENTERPRISE JAVA BEANS

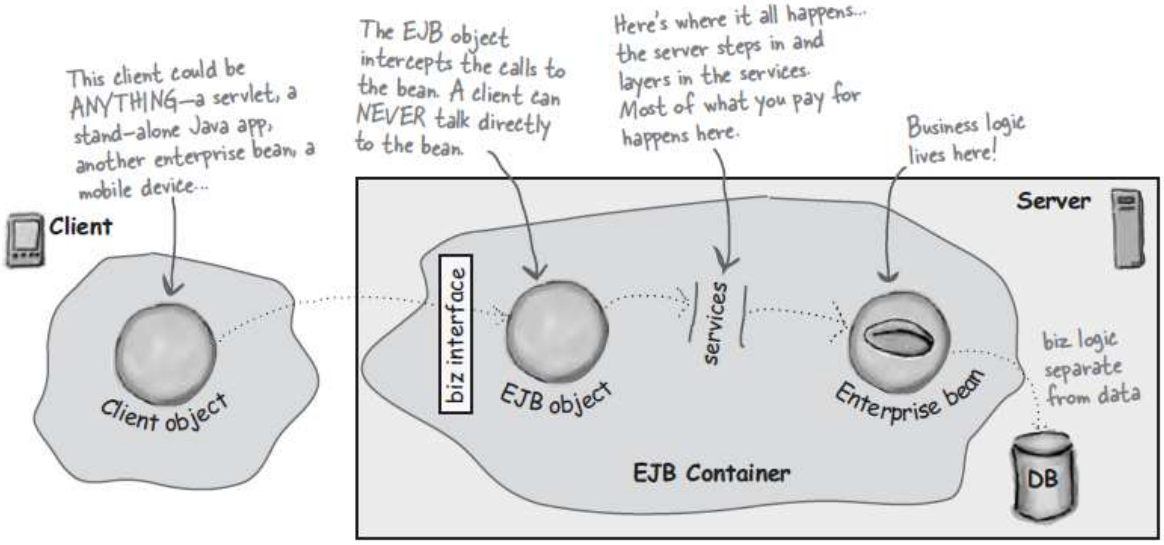
EJB stands for Enterprise Java Beans. EJB is an essential part of a J2EE platform. J2EE platform have component based architecture to provide multi-tiered, distributed and highly transactional features to enterprise level applications.

EJB provides an architecture to develop and deploy component based enterprise applications considering robustness, high scalability and high performance. An EJB application can be deployed on any of the application server compliant with J2EE 1.3 standard specification. Enterprise Bean is a server side component written in Java Language. It incorporates the business logic of an application (the code that implements the purpose of the application). Enterprise Bean solves problems of transactions, security, scalability, concurrency, communication, resource management, persistence, error handling and OS dependence as the user does not have to worry about them.

Benefits of EJB are

- Simplified development of large scale enterprise level application.
- Application Server/ EJB container provides most of the system level services like transaction handling, logging, load balancing, persistence mechanism, exception handling and so on. Developer has to focus only on business logic of the application.
- EJB container manages life cycle of EJB instances thus developer needs not to worry about when to create/delete EJB objects.
- EJB lets you customize and configure reusable components at deploy time without having to change the source code
- EJBs are portable, not just to different JVMs but also to different EJB servers.

A simple high level architecture of EJB is as follows:



- A client can be a servlet, stand-alone Java app, another bean or a mobile device that needs access to an EJB.
- The client access the EJB server through the Business Interface.
- The EJB object intercepts the request from the client. A client is never allowed direct access to the EJB.
- The EJB server checks the credentials of the client, provides any additional services which have to be provided and then the EJB is called

- The called EJB does the work and the response is sent back to the client.

EJB are primarily of three types which are briefly described below:

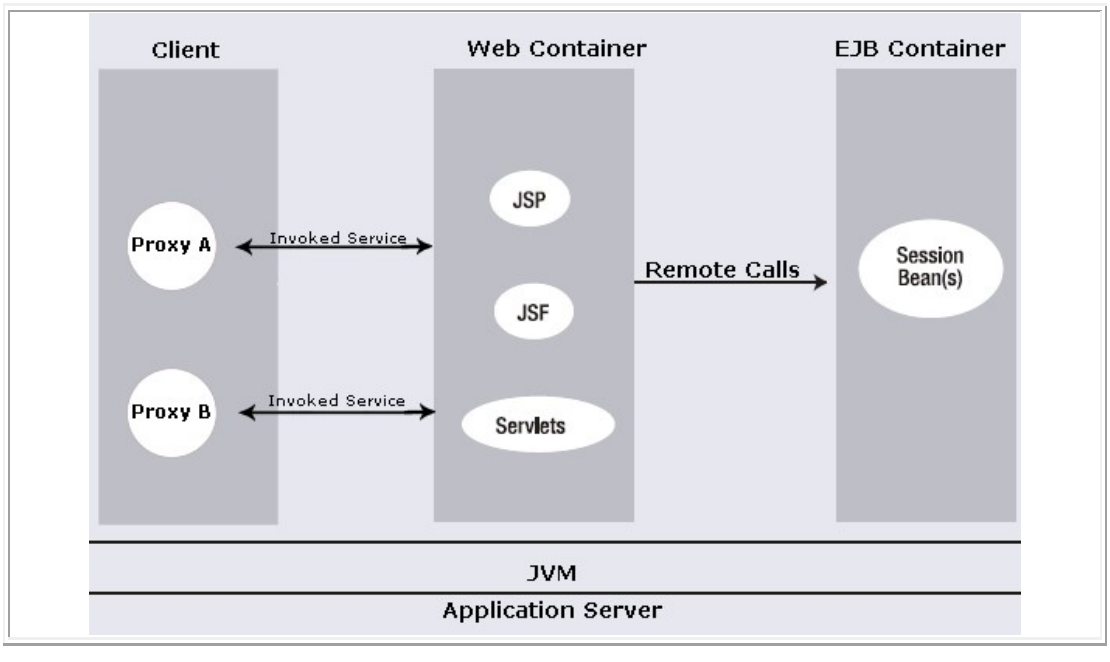
Type	Description
Session Bean	Session bean stores data of a particular user for a single session. It can be stateful or stateless. It is less resource intensive as compared to entity beans. Session bean gets destroyed as soon as user session terminates.
Entity Bean	Entity beans represent persistent data storage. User data can be saved to database via entity beans and later on can be retrieved from the database in the entity bean.
Message Driven Bean	Message driven beans are used in context of JMS (Java Messaging Service). Message Driven Beans can consumes JMS messages from external entities and act accordingly.

SESSION BEAN

What is a Session bean

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. An application may contain multiple sessions depending upon the number of users accessing to the application. A session bean makes an interactive session only for a single client and shields that client from complexities just by executing the business task on server side. 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. The session bean decides what data is to be modified. Typically, the session bean uses an entity bean to access or modify data. They 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.

A session bean can neither be shared nor can persist (means its value cannot be saved to the database) its value. A session bean can have only one client. As soon as the client terminates, the session bean associated with this client is also terminated and the data associated with this bean is also destroyed.



The above figure shows how Session Bean interacts with the clients as well as with the Entity Beans.

Session beans are divided into two parts.

- **Stateless Session Beans:** A stateless session bean does not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the instance of bean variables may contain a state specific to that client only for the duration of a method invocation. Once the method is finished, the client-specific state should not be retained i.e. the **EJB** container destroys a stateless session bean.

  These types of session beans do not use the class variables (instance variables). So they do not persist data across method invocation and therefore there is no need to passivates the bean's instance. Because stateless session beans can support multiple clients, they provide the better scalability for applications that require large numbers of clients.

- **Stateful Session Beans:** 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. The client sets the data to these variables which he wants to persist. A stateful session bean retains its state across multiple method invocations made by the same client. If the stateful session bean's state is changed during a method invocation, then that state will be available to the same client on the following invocation. 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**.

  For example, consider a customer using a debit card at an **ATM** machine. The ATM could perform various operations like checking an account balance, transferring funds, or making a withdrawal. These operations could be performed one by one, by the same customer. So the bean needs to keep track its state for each of these operations to the same client. Thus Stateful session beans has the extra overhead for the server to maintain the state than the stateless session bean.

The user interface calls methods of session beans if the user wants to use the functionality of the session bean. Session beans can call to other session beans and entity beans.

When to use session beans:

Generally session beans are used in the following circumstances:

- When there is only one client is accessing the beans instance at a given time.
- When the bean is not persistent that means the bean is going to exist no longer.
- The bean is implementing the web services.

Stateful session beans are useful in the following circumstances:

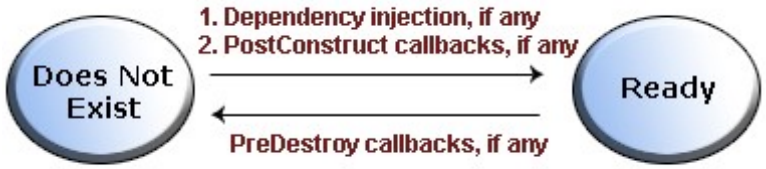
- What the bean wants to holds information about the client across method invocation.
- When the bean works as the mediator between the client and the other component of the application.
- When the bean have to manage the work flow of several other enterprise beans.

Stateless session beans are appropriate in the circumstances illustrated below:

- If the bean does not contain the data for a specific client.
- If there is only one method invocation among all the clients to perform the generic task.

Life Cycle of a Stateless Session Bean: Since the Stateless session bean does not passivate across method calls therefore a stateless session bean includes only two stages. Whether it does not exist or ready for

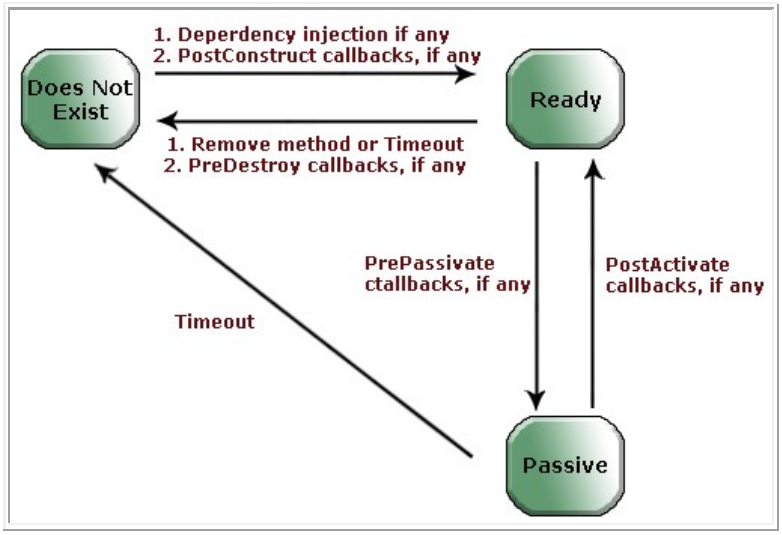
method invocation. A stateless session bean starts its life cycle when the client first obtains the reference of the session bean. For this, the container performs the dependency injection before invoking the annotated **@PreConstruct** method if any exists. After invoking the annotated @PreConstruct method the bean will be ready to invoke its method by the client.



The above figure demonstrates how the Stateless Session Beans are created and destroyed.

The container calls the annotated **@PreDestroy** method while ending the life cycle of the session bean. After this, the bean is ready for garbage collection.

Life Cycle of a Stateful Session Bean: 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 (Generally the container uses the least recently use algorithm to passivates a bean). 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.



The above image shows the various states of the Stateful Session Beans

While ending 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 BEAN

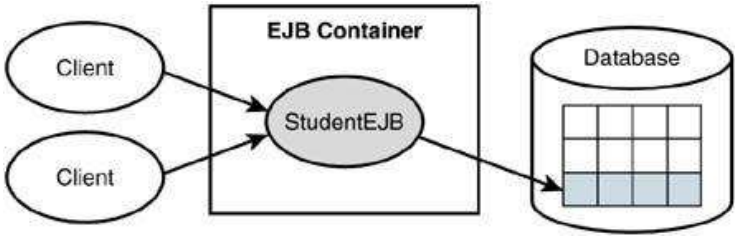
An entity bean is a server-side component that represents an object-oriented view of entities stored in persistent storage, such as a database, or entities that are implemented by an existing enterprise application. Entity beans are persistent objects. Persistence means that the state of the entity bean in memory is synchronized with the data it represents in the database. Entity beans typically contain data-related logic that performs a task such as inserting, updating, or removing a customer record in the database.

Characteristics of Entity Beans

Entity beans have the following characteristics:

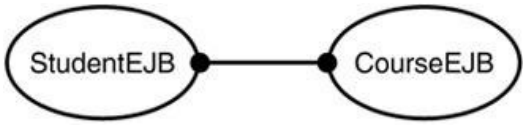
- Provides an object view of data in the database. For example, as shown in below figure the `StudentEJB` enterprise bean instance provides an object view of a record in the student table in the database. The EJB container transparently synchronizes the data between the server's memory and the database.

An entity bean provides an object view of data in the database.



- Represents long-lived data. Its lifetime is the same as data in the database. The data of entity beans survives a crash of the EJB container.
- Allows shared access by multiple users.
- Just as a relational database has the concept of a primary key, its primary key identifies an entity bean. A student entity bean, for example, might be identified by a `student ID`.
- An entity bean may be related to other entity beans. For example, as shown in below figure, in a college enrollment application, `StudentEJB` and `CourseEJB` would be related because students enroll in classes. This is an example of many-to-many bidirectional relationship.

An entity bean can be related to other entity beans.



COMPARING ENTITY BEANS AND SESSION BEANS

Entity beans differ from session beans in several ways. Entity beans are persistent, allow shared access, have primary keys, and may participate in relationships with other entity beans. The below table shows, the differences between session and entity beans.

Differences Between Session and Entity Beans	
Session Bean	**Entity Bean**
It contains business process logic, such as sending an e-mail, looking up the stock price from a database, implementing compression and encryption algorithms. They also implement business logic such as workflow, algorithms, and business rules.	It represents business entities, such as customers, products, accounts, and orders.
Executes a particular business task on behalf of a single client during a single session.	Can be shared by multiple clients.
The state is not persistent.	The state is persistent.
Usually does not correspond directly to data.	Usually corresponds directly to data.

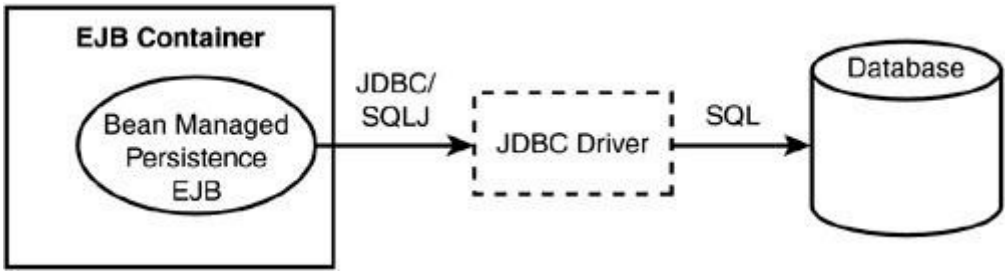
TYPES OF ENTITY PERSISTENCE/BEANS

Entity beans have two types of persistence: bean-managed persistence (BMP) and container-managed persistence (CMP). These two types are based on who is responsible for entity bean persistence bean provider or EJB provider.

Bean-Managed Persistence

With bean-managed persistence, the entity bean code that we write contains the calls that access the database. The data access can be coded into the entity bean class or encapsulated in a data access object that is part of the entity bean. The below figure how bean-managed persistence works with database access.

Bean-managed persistence contains the calls that access the database.



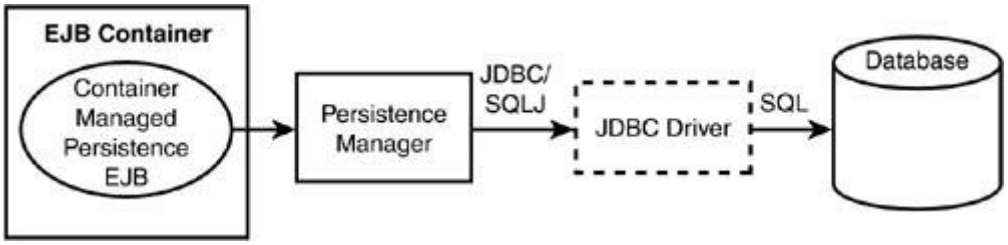
For example, as shown in the following snippet, the `StudentEJB` method `ejbCreate` contains the JBDC code to insert a student record into the `students` table:

```
public class StudentEJB implements EntityBean {

  private String studentId; // also the primary Key
  private String firstName;
  private String lastName;
  private String address;

  public String ejbCreate(String studentId, String firstName,
    String lastName, String address) throws CreateException
  {
    this.studentId = studentId;
    this.firstName = firstName;
    this.lastName = lastName;
    this.address = address;

    Connection con = null;
    PreparedStatement ps = null;

    try {
      con = getConnection();
      ps = con.prepareStatement("insert into students (student_id, "+
        " first_name, last_name, address) values (?, ?, ?, ?) ");
      ps.setString(1, studentId);
      ps.setString(1, firstName);
      ps.setString(1, lastName);
      ps.setString(1, address);
      if (ps.executeUpdate() != 1) {
        String error = "JDBC did not create any row";
        log(error);
        throw new CreateException (error);
      } return studentId;} catch (SQLException sqe) {
    ...
    } finally { cleanup(con, ps);    }   }   ... }
```

Container-Managed Persistence

If our bean has container-managed persistence (CMP), the EJB container uses a persistence manager and automatically generates the necessary database access calls. The code write for the entity bean does not include these calls. The persistence manager is responsible for persistence of the entity bean, including creating, loading, and removing the entity bean instance in the database.

The persistence manager is responsible for persistence of CMP.



For example, as shown in the following listing, the `OrderEJB` method `ejbCreate` does not contain any JBDC code to insert order records into the `orders` table:

```
public abstract class OrderEJB implements EntityBean {

       /* get and set methods for cmp fields */
        public abstract String getOrderId();
        public abstract void setOrderId(String id);

        public abstract java.sql.Timestamp getOrderDate();
        public abstract void setOrderDate(java.sql.Timestamp timestamp);

        public abstract String getStatus();
        public abstract void setStatus(String status);

        public abstract double getAmount();
        public abstract void setAmount(double amount);

       /* get and set methods for relationship fields */
        public abstract Collection getLineItems();
        public abstract void setLineItems(Collection lineItems);

        public abstract StudentLocal getStudent();
        public abstract void setStudent(StudentLocal student);

        public String ejbCreate(String orderId, StudentLocal student,
          String status,double amount) throws CreateException {
         setOrderId(orderId);
         setOrderDate(new java.sql.Timestamp(System.currentTimeMillis()));
         setStatus(status);
         setAmount(amount);

         return null;
        }
  ...
}
```

We have to specify the persistence fields, relationship fields, schema, and queries in the deployment descriptor. Based on this information, the container tools generate data access calls corresponding to the underlying database, at deployment time.

Persistent Fields: A persistent field is designed to represent and store a single unit of data. Collectively, these fields constitute the state of the bean. During deployment, the container maps the entity bean to a database table and maps the persistent fields to the table's columns. At runtime, the EJB container automatically synchronizes this state with the database.

An `OrderEJB` entity bean, for example, might have persistent fields such as `orderId`, `orderDate`, `status`, and `amount`. In container-managed persistence, these fields are virtual. We declare them in the abstract schema, but do not code them as instance variables in the entity bean class. Instead, the persistent fields are identified in the code by access methods (getters and setters).

Relationship Fields: A relationship field is designed to represent and store a reference to another entity bean. This is analogous to a foreign key in a relational database table. Unlike a persistent field, a relationship field does not constitute the state of an entity bean.

An `OrderEJB` entity bean, for example, might have relationship fields such as `lineItems` and `student`.

Abstract Persistent Schema: The CMP entity bean's deployment descriptor contains a description of the bean's abstract persistent schema. This schema is an abstract representation of an entity bean's persistent fields and relationship fields. The abstract schema is independent of the entity bean's implementation in a particular EJB container or particular database.

For example, the following snippet declares an abstract persistent schema named `Order` in the deployment descriptor:

```
. . .
<cmp-version>2.x</cmp-version>
<abstract-schema-name>Order</abstract-schema-name>
<cmp-field>
   <field-name>orderId</field-name>
</cmp-field>
 <cmp-field>
   <field-name>orderDate</field-name>
 </cmp-field>
 <cmp-field>
   <field-name>status</field-name>
 </cmp-field>
 <cmp-field>
   <field-name>amount</field-name>
 </cmp-field>
 . . .
```

The `cmp-version` must be 2.x if we want to take advantage of EJB 2.0 container-managed persistence. The schema declares four container-managed persistent fields (`cmp-field`): `orderId`, `orderDate`, `status`, and `amount`. The names of these fields must match the abstract get and set methods in your entity bean class. For example, your entity bean class's methods must be `getOrderId`, `setOrderId`, `getOrderDate`, `setOrderDate`, `getStatus`, `setStatus`, `getAmount`, and `setAmount`.

The EJB Query Language: The EJB Query language (EJB QL) is new in EJB 2.0. EJB QL is similar to SQL. It enables you to specify queries of entity bean methods in a database-independent, portable way. For example, the following code line illustrates how the application uses an EJB QL query of the method `findOrdersByStatus(String status)` to find orders of a particular status, such as `COMPLETE`:

```
SELECT OBJECT(o) FROM Order AS o WHERE o.status = ?1
```

The preceding query returns `Order` objects, as indicated by the expression `OBJECT(o)`. The identifier `o` is analogous to an SQL correlation variable. The `WHERE` clause limits the orders to those whose status matches the value of the first parameter passed to the query, which is denoted by the expression `?1`.

The container tools will translate such queries into the target language of the underlying database. For example, in case of a relational database, the container translates an EJB QL query into an SQL query.

Note: Entity beans with container-managed persistence are portable. Their code is not tied to a specific database.

When to Use BMP or CMP

The choice between BMP and CMP is often decided by factors such as portability, availability of container tools, flexibility, and so on.

Using BMP: We would use bean-managed persistence in the following situations:

- Complete control of managing persistence, such as writing optimized queries.
- Writing persistence logic to a very proprietary legacy database system for which container tools do not exist.
- Persistent store is not a database.

Using CMP: In container managed persistence the container takes responsibility for generating the data access code. This simplifies the task of writing entity beans.

The CMP entity bean's code is not tied to a specific persistent storage mechanism (database). Because of this flexibility, even if redeploy the same entity bean on different J2EE servers that use different databases, there is no need to modify or recompile the bean's code. In short, your CMP entity beans are more portable than BMP entity beans.

INSTANCE POOL AND INSTANCE CACHE

An EJB container maintain an instance pool of each type of entity bean. This saves the precious time of creating and destroying of objects. At startup, the container creates instances as specified in the deployment descriptor of the entity bean. While the instance is in the available pool, the instance is not associated with any particular entity object identity. All instances in the pool are considered equivalent; therefore, an instance can be assigned by the container to any entity object identity. We can control the instance pool size in the vendor-specific deployment descriptor.

For example, we can specify the pool size in the deployment descriptor `weblogic-ejb-jar.xml` for WebLogic Server as follows:

```
<pool>
  <max-beans-in-free-pool>100</max-beans-in-free-pool>
  <initial-beans-in-free-pool>50</initial-beans-in-free-pool>
</pool>
```

Similarly, we can specify the pool size in the deployment descriptor `jboss.xml` for the JBoss server as follows:

```
<instance-pool>org.jboss.ejb.plugins.EntityInstancePool</instance-pool>
<container-pool-conf>
  <MaximumSize>100</MaximumSize>
  <MinimumSize>10</MinimumSize>
</container-pool-conf>
```

The EJB container can have an instance cache to manage all entity bean instances that are associated with an identity. In large applications, the number of clients connected concurrently to a Web site can be in the thousands. This can have an adverse effect on performance when resources are used up. Passivation and activation are mechanisms provided by the EJB container to manage valuable resources, such as memory, in order to reduce the number of entity bean instances needed to service all concurrent clients.

Passivation is the mechanism by which the EJB container stores the bean's state into the database. The container starts passivation as soon as the number of allocated entity beans exceeds a certain threshold. The EJB container provides the bean developer with `ejbPassivate()` as a callback method to release any allocated resources. ***Activation***, on the other hand, is the process of restoring back the bean from the database. The EJB container activates a passivated instance when the bean's client references the bean

instance. The EJB container provides the bean developer with `ejbActivate()` as a callback method to restore any connections and other resources.

For example, we can specify the instance cache size in deployment descriptor `weblogic-ejb-jar.xml` for WebLogic Server as follows:

```
<entity-cache>
 <max-beans-in-cache>1000</max-beans-in-cache>
</entity-cache>
```

Similarly, we can specify the instance cache size in deployment descriptor `jboss.xml` for JBoss server as follows:

```
<instance-cache>
<container-cache-conf>
 <cache-policy>
  <cache-policy-conf>
    <min-capacity>5</min-capacity>
    <max-capacity>10</max-capacity>
  </cache-policy-conf>
 </cache-policy>
</container-cache-conf>
```

ENTITY BEAN FILES

An entity bean consists of a home interface, component interface, enterprise bean class, and deployment descriptor. In addition, an entity bean can have a primary key class.

In most cases, primary key class will be a `String` or an `Integer`, which belong to J2SE standard libraries. For example, the primary key class for `StudentEJB` is `studentId`, which is a `String`.

For some entity beans, we must define our own primary key class. For example, if our primary key is composed of multiple fields (a composite primary key), define our own primary key class.

```
</instance-cache>
```

Note: Instance pooling is used to manage EJB instances that are not associated with any identity. Instance caching is used to manage EJB instances that are associated with an identity. Instance pooling is applicable to stateless session, entity, and message-driven beans, whereas instance caching is applicable to stateful session and entity beans.

Entity Bean Methods

The entity bean class implements the `javax.ejb.EntityBean` interface. Therefore need to implement the `setEntityContext`, `unsetEntityContext`, `ejbActivate`, `ejbPassivate`, `ejbLoad`, `ejbStore`, and `ejbRemove` methods defined in the `javax.ejb.EntityBean` interface. The bean may implement `ejbCreate<method>(...)` methods (and corresponding `ejbPostCreate<method>(...)` methods) used to initialize the bean instance. In addition, the bean class can implement business methods, home methods and remove methods.

The below table summarizes the entity bean methods.

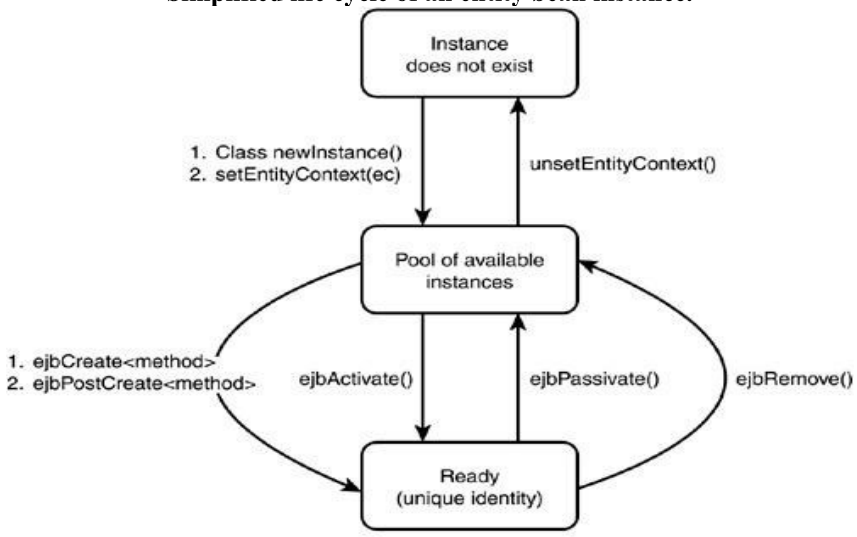
Method	Purpose	What You Need to Do
`SetEntityContext (EntityContext)`	The EJB container calls this method to set the associated entity context.	Store the reference to the entity context object in an instance variable, if you need it later. You also allocate any resources that are to be held for the lifetime of the instance.
`unsetEntityContext()`	The container invokes this method before terminating the life of the instance.	Free any resources that are held by the instance.
`ejbCreate<method>(...)`	The EJB container invokes the corresponding `ejbCreate<method>` method when a client invokes a `create<method>` method on the home interface.	Each entity class can have zero or more `ejbCreate<method>(...)` methods and each one can take different arguments. In BMP, validate the client-supplied parameters and insert a record into the database. The method also initializes the instance's variables. In CMP, validate the client-supplied parameters and initialize the enterprise bean state.
`ejbPostCreate<method>(...)`	The container invokes the matching `ejbPostCreate<method>(...)` method on an entity instance after it invokes the `ejbCreate<method>(...)` method with the same arguments.	For each `ejbCreate<method>(...)` method, you must have a matching `ejbPostCreate<method>(...)` method. This method enables you to complete any remaining initialization of entity bean instances.
`ejbActivate()`	The EJB container calls this method when it picks the entity instance from the instance pools and associates it to a specific entity object identity.	Acquire any resources needed to service a particular client; for example, open socket connections.
`ejbPassivate()`	The EJB container calls this method when the container decides to disassociate the instance from an entity object identity and to put it back into the instance pool.	Release any resources that you acquired to service a particular client; for example, close socket connections.
Business methods	The business methods contain business logic that you want to encapsulate within your entity bean.	Write business logic in these methods.

`ejbLoad()`	The EJB container invokes this method to instruct the instance to synchronize its state by loading it from the underlying database.	In BMP, refresh the instance variables by reading from the database. Also recalculate any dependent values. In CMP, recalculate the values of any instance variables that depend on the persistent fields; for example, transient fields.
`ejbStore()`	The EJB container invokes this method to instruct the instance to synchronize its state by storing it to the underlying database.	In BMP, write any updates cached in the instance variables to the database. In CMP, prepare the container-managed fields to be written to the database.
`ejbFind<method>(...)`	Finder methods allow clients to locate entity beans	In BMP, for every finder method defined in the home interface, you must implement a corresponding method that begins with the `ejbFind` prefix. With CMP, you do not write the `ejbFind` methods in your entity bean class.
`ejbHome<method>(...)`	Home methods contain business logic that that is not specific to an entity bean instance.	Implement the business logic using other methods or JDBC code.
`ejbRemove()`	The container calls this method as a result of the client's invocation of a remove method.	In BMP, remove the entity state from the database and release any resources that you acquired to service a particular client. With CMP, release any resources that you acquired to service a particular client.

Life Cycle of an Entity Bean

The below figure shows a simplified state diagram of an entity bean instance.

Simplified life cycle of an entity bean instance.



Life cycle of an entity bean instance

Initially, the bean instance does not exist.

A Bean instance's life cycle starts when the container creates the instance using `Class.newInstance()` and then calls the `setEntityContext` method. Now the instance enters a pool of available instances. An instance in the pooled state is not associated with any particular entity object identity. All instances in the pooled state are identical. While the instance is in the pooled state, the EJB container may use the instance to execute any of the entity bean's finder methods or home methods.

Our Bean instance moves from the pooled state to the ready state when the container selects that instance to service a client call to an entity object. There are two paths from the pooled stage to the ready stage. On the first path, the client invokes the `create` method, causing the EJB container to call the `ejbCreate` and `ejbPostCreate` methods. On the second path, the EJB container invokes the `ejbActivate` method.

While in the ready state, the instance is associated with a specific entity object identity. The container calls business methods on the instance, based on the client call. The EJB container also can synchronize the state of the instance with the database using methods `ejbLoad` and `ejbStore`.

Eventually, the EJB container will transition the instance to the pooled state. This happens when the client calls the `remove` method, which causes the EJB container to call the `ejbRemove` method. Second, the EJB container might call the `ejbPassivate` method.

At the end of the instance's life cycle, the EJB container removes the instance from the pool and invokes the `unsetEntityContext` method.

Note : In bean-managed persistence, when the EJB container moves an instance from the pooled state to the ready state, it does not automatically set the primary key. Therefore, the `ejbCreate` and `ejbActivate` methods must set the primary key.

MESSAGE-DRIVEN BEAN

A **message-driven bean** is an enterprise bean that allows Java EE applications to process messages asynchronously. It normally acts as a JMS message listener, which is similar to an event listener except that it receives JMS messages instead of events. The messages can be sent by any Java EE component (an application client, another enterprise bean, or a web component) or by a JMS application or system that does not use Java EE technology. Message-driven beans can process JMS messages or other kinds of messages.

The most visible difference between message-driven beans and session beans is that clients do not access message-driven beans through interfaces. Unlike a session bean, a message-driven bean has only a bean class.

In several respects, a message-driven bean resembles a stateless session bean.

- A message-driven bean's instances retain no data or conversational state for a specific client.
- All instances of a message-driven bean are equivalent, allowing the EJB container to assign a message to any message-driven bean instance. The container can pool these instances to allow streams of messages to be processed concurrently.
- A single message-driven bean can process messages from multiple clients.

The instance variables of the message-driven bean instance can contain some state across the handling of client messages (for example, a JMS API connection, an open database connection, or an object reference to an enterprise bean object).

Client components do not locate message-driven beans and invoke methods directly on them. Instead, a client accesses a message-driven bean through, for example, JMS by sending messages to the message

destination for which the message-driven bean class is the MessageListener. We assign a message-driven bean's destination during deployment by using Application Server resources.

Message-driven beans have the following characteristics:

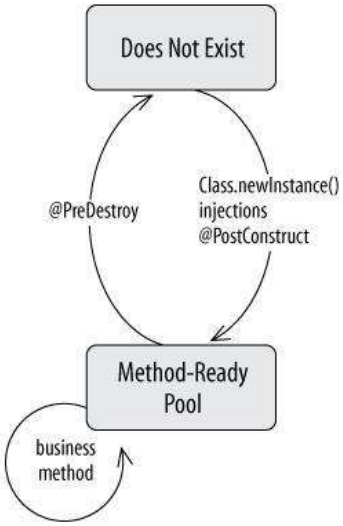
- They execute upon receipt of a single client message.
- They are invoked asynchronously.
- They are relatively short-lived.
- They do not represent directly shared data in the database, but they can access and update this data.
- They can be transaction-aware.
- They are stateless.

When a message arrives, the container calls the message-driven bean's **onMessage** method to process the message. The **onMessage** method normally casts the message to one of the five JMS message types and handles it in accordance with the application's business logic. The **onMessage** method can call helper methods, or it can invoke a session bean to process the information in the message or to store it in a database.

A message can be delivered to a message-driven bean within a transaction context, so all operations within the **onMessage** method are part of a single transaction. If message processing is rolled back, the message will be redelivered.

Use of Message-Driven Beans

Just as session beans have well-defined lifecycles, so does the message-driven bean. The MDB instance's lifecycle has two states: ***Does Not Exist*** and ***Method-Ready Pool***. The Method-Ready Pool is similar to the instance pool used for stateless session beans. The following diagram illustrates the lifecycle of a Message Driven Bean.



The Does Not Exist State: When an MDB instance is in the Does Not Exist state, it is not an instance in the memory of the system. In other words, it has not been instantiated yet.

The Method-Ready Pool: MDB instances enter the Method-Ready Pool as the container needs them. When the EJB server is first started, it may create a number of MDB instances and enter them into the Method-Ready Pool (the actual behavior of the server depends on the implementation). When the number of MDB instances handling incoming messages is insufficient, more can be created and added to the pool.

Transitioning to the Method-Ready Pool: When an instance transitions from the Does Not Exist state to the Method-Ready Pool, three operations are performed on it. First, the bean instance is instantiated by invoking the ***Class.newInstance( )*** method on the bean implementation class. Second, the container injects

any resources that the bean's metadata has requested via an injection annotation or XML deployment descriptor.

Finally, the EJB container will invoke the PostConstruct callback if there is one. The bean class may or may not have a method that is annotated with @javax.ejb.PostConstruct. If it is present, the container will call this annotated method after the bean is instantiated. This @PostConstruct annotated method can be of any name and visibility, but it must return void, have no parameters, and throw no checked exceptions. The bean class may define only one @PostConstruct method (but it is not required to do so).

```
@MessageDriven
public class MyBean implements MessageListener {

  @PostConstruct
  public void myInit() {}
```

MDBs are not subject to activation, so they can maintain open connections to resources for their entire lifecycles. The @PreDestroy method should close any open resources before the stateless session bean is evicted from memory at the end of its lifecycle.

Life in the Method-Ready Pool: When a message is delivered to an MDB, it is delegated to any available instance in the Method-Ready Pool. While the instance is executing the request, it is unavailable to process other messages. The MDB can handle many messages simultaneously, delegating the responsibility of handling each message to a different MDB instance. When a message is delegated to an instance by the container, the MDB instance's MessageDrivenContext changes to reflect the new transaction context. Once the instance has finished, it is immediately available to handle a new message.

Transitioning out of the Method-Ready Pool: The death of an MDB instance: Bean instances leave the Method-Ready Pool for the Does Not Exist state when the server no longer needs them—that is, when the server decides to reduce the total size of the Method-Ready Pool by evicting one or more instances from memory. The process begins by invoking an @PreDestroy callback method on the bean instance. Again, as with @PostConstruct, this callback method is optional to implement and its signature must return a void type, have zero parameters, and throw no checked exceptions. A @PreDestroy callback method can perform any cleanup operation, such as closing open resources.

```
@MessageDriven
public class MyBean implements MessageListener {

  @PreDestroy
  public void cleanup( ) {
    ...
  }
```

As with @PostConstruct, @PreDestroy is invoked only once: when the bean is about to transition to the Does Not Exist state. During this callback method, the MessageDrivenContext and access to the JNDI ENC are still available to the bean instance. Following the execution of the @PreDestroy method, the bean is dereferenced and eventually garbage-collected.

DIFFERENCES BETWEEN SESSION BEAN AND ENTITY BEAN

Session bean	Entity bean
Represents a single conversation with a client. Typically, encapsulates an action or actions to be taken on business data.	Typically, encapsulates persistent business data—for example, a row in a database.

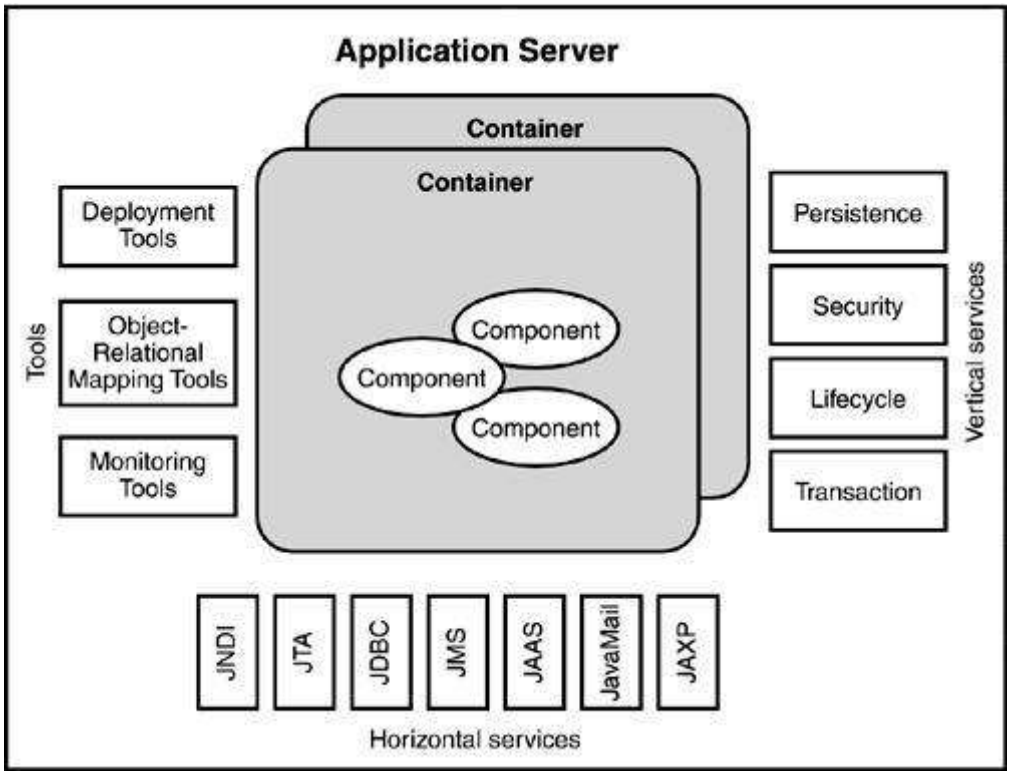
Session bean	Entity bean
Is relatively short-lived.	Is relatively long-lived.
Is created and used by a single client.	May be shared by multiple clients.
Has no primary key.	Has a primary key, which enables an instance to be found and shared by more than one client.
Typically, persists only for the life of the conversation with the client. (However, may choose to save information.)	Persists beyond the life of a client instance. Persistence can be container-managed or bean-managed.
Is not recoverable—if the EJB server fails, it may be destroyed.	Is recoverable—it survives failures of the EJB server.
May be stateful (that is, have a client-specific state) or stateless (have no non-transient state).	Is typically stateful.
May or may not be transactional. If transactional, can manage its own OTS transactions, or use container-managed transactions. A stateful session bean that manages its own transactions can begin an OTS transaction in one method and commit or roll it back in a subsequent method. A stateless session bean that manages its own transactions and begins an OTS transaction must commit (or roll back) the transaction in the same method in which it was started. The state of a transactional, stateful session bean is not automatically rolled back on transaction rollback. In some cases, the bean can use session synchronization to react to syncpoint.	May or may not be transactional. Must use the container-managed transaction model. If transactional, its state is automatically rolled back on transaction rollback.
Is not re-entrant.	May be re-entrant.

EJB CONTAINERS

An EJB container is an abstract facility that manages instances of EJB components. The EJB specification defines the contractual agreement between the EJB and its container to provide both infrastructure and runtime services. Clients never access beans directly. Access is gained through container-generated methods, which in turn invoke the beans methods. A container vendor may also provide additional services implemented in either the container or the server.

Essential EJB Container Services

All EJB instances run within an EJB container. The container provides system-level services to its EJBs and controls their life cycle. Because the container handles most system-level issues, the EJB developer does not have to include this logic with the business methods of the enterprise bean. In general, J2EE containers provide three main types of services: common vertical services, common horizontal services, and common deployment tools shown in below figure.



1. **Common Vertical Services:** The common vertical services are inherent services that are provided by the EJB container and are not specified explicitly by the J2EE architecture APIs. They contribute to the performance and runtime aspects of the EJBs and the services provided to them. EJB developers need not include any logic to manage these services. The following is a list of these common services:
   - **Life cycle management:** The container creates and destroys enterprise bean instances based on client demand. This is done to maximize performance and minimize resource usage such as memory. In addition, a container may transparently multiplex a pool of instances to share among several clients.
   - **Security:** The security services are designed to ensure that only authorized users access resources. J2EE specifies a simple role-based security model for enterprise beans and Web components. In addition, vendors typically provide integration with third-party security providers such as LDAP.
   - **Remote method invocation:** The container transparently manages the communication between enterprise beans and other components. So, we do not have to worry about low-level communication issues such as initiating the connections, and marshalling/unmarshalling the method parameters. A bean developer simply writes the business methods as if they will be invoked on a local platform.
   - **Transaction management:** The transaction services relieve the enterprise bean developer from dealing with the complex issues of managing distributed transactions that span multiple enterprise beans and resources such as databases. The container ensures that updates to all the databases occur successfully; otherwise, it rolls back all aspects of the transaction.
   - **Persistence:** Persistence services simplify the connection between the application and database tiers. Container-managed persistence of entity beans simplifies the coding effort for application developers.
   - **Passivation/activation:** The mechanism that is used by the container to store an inactive enterprise bean to disk, and restore its state when the bean is invoked. The container uses this mechanism in support of both entity and stateful session beans. This allows the servicing of more active clients by dynamically freeing critical resources such as memory.
   - **Clustering:** Supports replication of EJBs and services across multiple application server instances installed on the same machine or in different environments. Clustering involves load-balancing the

requested services and EJBs among the replicated instances. It also supports fail-over—should one instance fail, the load will be picked up by another.

- **Concurrency:** Supports multithreading management. All components must be developed as single-threaded, and the container manages the concurrency and serialization access to the shared resources.
- **Resource pooling:** Supports the allocation of a pool of instances, and then assigns them to the requesting clients. When an instance is free, it goes back to the pool. This is applied for JDBC connections, stateless session beans, and entity beans.

2. **Common Horizontal Services:** Common horizontal services are the services specified in the J2EE architecture. They're commonly known as J2EE APIs, and are provided by the EJB server to all the containers running on the server. Here's the standard list of the J2EE APIs:

- **Java Naming and Directory Interface (JNDI):** Provides Java-technology-enabled applications with a unified interface to multiple naming and directory services in the enterprise.
- **Java Database Connectivity (JDBC):** Provides access to virtually any tabular data source for J2EE applications.
- **JavaServer Pages (JSP):** Enables Web developers and designers to rapidly develop and easily maintain information-rich, dynamic Web pages that leverage existing business systems.
- **Java Servlet:** Provides Web developers with a simple, consistent mechanism for extending the functionality of a Web server and for accessing existing business systems.
- **Java Transaction API (JTA):** The standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the resource manager, the application server, and the transactional applications.
- **Java Message Service (JMS):** A common API and provider framework that enables the development of portable, message-based enterprise applications.
- **J2EE Connector Architecture (JCA):** The key component for enterprise application integration in the Java platform. In addition to facilitating enterprise application integration, the JCA helps to integrate existing enterprise applications and information systems with Web services and applications.
- **Java API for XML Processing (JAXP):** Enables applications to parse and transform XML documents independent of a particular XML processing implementation.
- **RMI over IIOP (RMI/IIOP):** Delivers Common Object Request Broker Architecture (CORBA) distributed computing capabilities to the J2EE platform.
- **Java Authentication and Authorization Security (JAAS):** Enables services to authenticate and enforce access controls upon users.
- **JavaMail:** Provides a platform- and protocol-independent framework for building Java-technology-based mail and messaging applications.
- **JavaBean Activation Framework (JAF):** Standard services used by JavaMail to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and instantiate the appropriate bean to perform said operation.

JAVA MESSAGING SYSTEM(JMS)

INTRODUCTION:

Messaging: Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages

from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is **loosely coupled**. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only which message format and which destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.

Messaging also differs from electronic mail (email), which is a method of communication between people or between software applications and people. Messaging is used for communication between software applications or software components.

JMS API:

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain.

The JMS API enables communication that is not only loosely coupled but also:

- **Asynchronous**: A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
- **Reliable**: The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

Use of JMS API:

An enterprise application provider is to choose a messaging API over a tightly coupled API, such as a remote procedure call (RPC), under the following circumstances.

- The provider wants the components not to depend on information about other components' interfaces, so components can be easily replaced.
- The provider wants the application to run whether or not all components are up and running simultaneously.
- The application business model allows a component to send information to another and to continue to operate without receiving an immediate response.

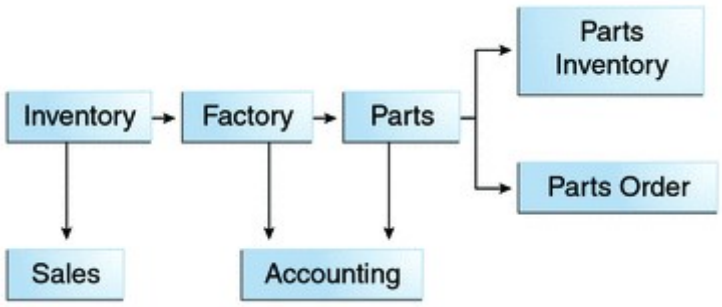
For example, components of an enterprise application for an automobile manufacturer can use the JMS API in situations like these:

- The inventory component can send a message to the factory component when the inventory level for a product goes below a certain level so the factory can make more cars.
- The factory component can send a message to the parts components so the factory can assemble the parts it needs.

- The parts components in turn can send messages to their own inventory and order components to update their inventories and to order new parts from suppliers.
- Both the factory and the parts components can send messages to the accounting component to update budget numbers.
- The business can publish updated catalogue items to its sales force.

Using messaging for these tasks allows the various components to interact with one another efficiently, without tying up network or other resources. The below figure illustrates how this simple example work.

Messaging in an Enterprise Application



Manufacturing is only one example of how an enterprise can use the JMS API. Retail applications, financial services applications, health services applications, and many others can make use of messaging.

JMS API Work with the Java EE Platform

When the JMS API was introduced in 1998, its most important purpose was to allow Java applications to access existing messaging-oriented middleware (MOM) systems, such as MQSeries from IBM. Since that time, many vendors have adopted and implemented the JMS API, so a JMS product can now provide a complete messaging capability for an enterprise.

Beginning with the 1.3 release of the Java EE platform, the JMS API has been an integral part of the platform, and application developers have been able to use messaging with Java EE components.

The JMS API in the Java EE platform has the following features.

- Application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message. Application clients can in addition receive JMS messages asynchronously. (Applets, however, are not required to support the JMS API.)
- Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages. A JMS provider can optionally implement concurrent processing of messages by message-driven beans.
- Message send and receive operations can participate in distributed transactions, which allow JMS operations and database accesses to take place within a single transaction.

The JMS provider can be integrated with the application server using the Java EE Connector architecture.

JMS API Architecture

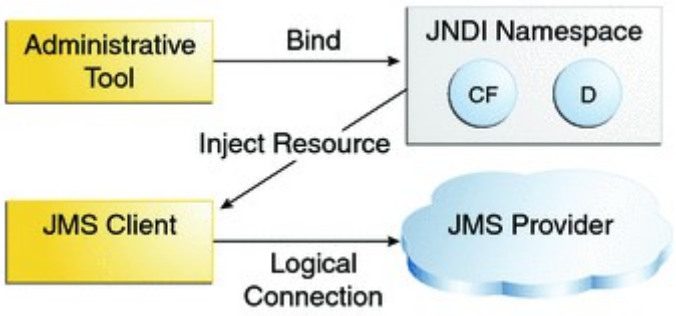
A JMS application is composed of the following parts.

- A **JMS provider** is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the Java EE platform includes a JMS provider.

- **JMS clients** are the programs or components, written in the Java programming language, that produce and consume messages. Any Java EE application component can act as a JMS client.
- **Messages** are the objects that communicate information between JMS clients.
- **Administered objects** are preconfigured JMS objects created by an administrator for the use of clients. The two kinds of JMS administered objects are destinations and connection factories.

The below figure illustrates the way these parts interact. Administrative tools allow you to bind destinations and connection factories into a JNDI namespace. A JMS client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.

JMS API Architecture



MESSAGING DOMAINS

The JMS API support both the point-to-point and the publish/subscribe domains, and some JMS clients combine the use of both domains in a single application. In this way, the JMS API has extended the power and flexibility of messaging products.

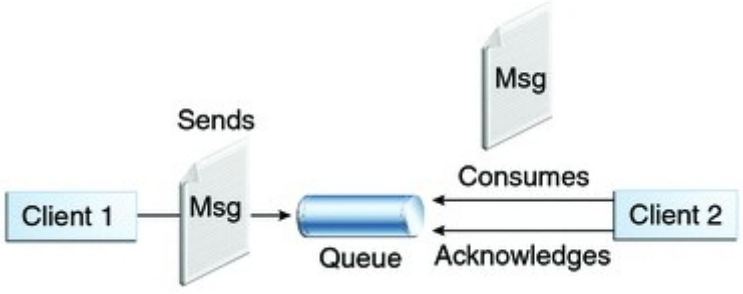
The JMS specification goes one step further: It provides common interfaces that enable us to use the JMS API in a way that is not specific to either domain.

Point-to-Point(PTP) Messaging Domain : A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages sent to them until the messages are consumed or expire.

PTP messaging, illustrated in below figure has the following characteristics:

- Each message has only one consumer.
- A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
- The receiver acknowledges the successful processing of a message.

Point-to-Point Messaging



Use PTP messaging when every message we send must be processed successfully by one consumer.

Publish/Subscribe Messaging Domain. In a publish/subscribe (pub/sub) product or application, clients address messages to a topic, which functions somewhat like a bulletin board. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.

Publish/subscribe messaging has the following characteristics.

- Each message can have multiple consumers.
- Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

Use Publish/subscribe messaging when each message can be processed by any number of consumers (or none).

The below figure illustrates Publish/subscribe messaging.

Publish/Subscribe Messaging

Programming with the Common Interfaces

Version 1.1 of the JMS API allows us to use the same code to send and receive messages under either the PTP or the Publish/subscribe domain. The destinations we use remain domain-specific, and the behavior of the application will depend in part on whether you are using a queue or a topic. However, the code itself can be common to both domains, making your applications flexible and reusable. This tutorial describes and illustrates these common interfaces.

Message Consumption

Messaging products are inherently asynchronous: There is no fundamental timing dependency between the production and the consumption of a message. However, the JMS specification uses this term in a more precise sense. Messages can be consumed in either of two ways:

- **Synchronously**: A subscriber or a receiver explicitly fetches the message from the destination by calling the `receive` method. The `receive` method can block until a message arrives or can time out if a message does not arrive within a specified time limit.
- **Asynchronously**: A client can register a **message listener** with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage` method, which acts on the contents of the message.

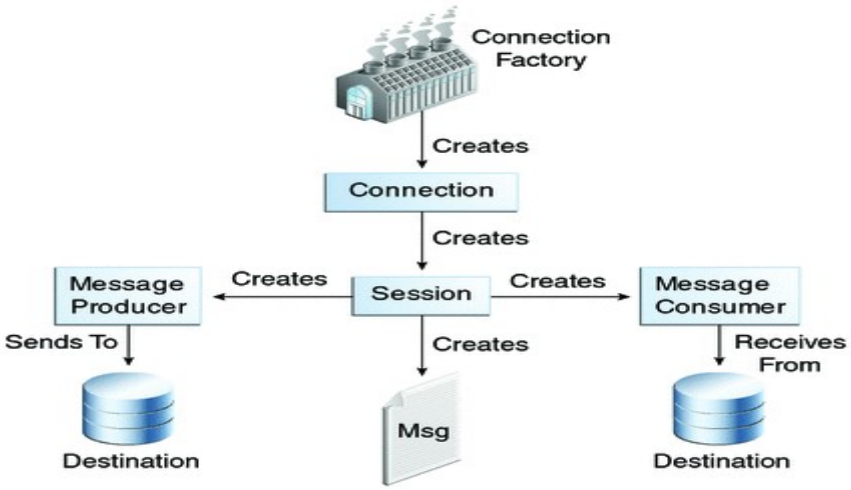
The JMS API Programming Model

The basic building blocks of a JMS application are:

- Administered objects: connection factories and destinations
- Connections
- Sessions
- Message producers
- Message consumers
- Messages

The figure shows how all these objects fit together in a JMS client application.

The JMS API Programming Model



- **JMS Administered Objects:** Two parts of a JMS application, destinations and connection factories, are best maintained administratively rather than programmatically..

  JMS clients access these objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of the JMS API. Ordinarily, an administrator configures administered objects in a JNDI namespace, and JMS clients then access them by using resource injection.

  - **JMS Connection Factories:** A connection factory is the object a client uses to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator. Each connection factory is an instance of the ConnectionFactory, QueueConnectionFactory, or TopicConnectionFactory interface.

    At the beginning of a JMS client program, we inject a connection factory resource into a ConnectionFactory object. For example, the following code fragment specifies a resource whose JNDI name is jms/ConnectionFactory and assigns it to a ConnectionFactory object:

    ```
    @Resource(lookup = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    ```

    In a Java EE application, JMS administered objects are normally placed in the jms naming subcontext.

- **JMS Destinations**

  A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging domain, destinations are called queues. In the pub/sub messaging domain, destinations are called topics. A JMS application can use multiple queues or topics (or both).

  In addition to injecting a connection factory resource into a client program, we usually inject a destination resource. Unlike connection factories, destinations are specific to one domain or the other. To create an application that allows us to use the same code for both topics and queues, you assign the destination to a `Destination` object.

  The following code specifies two resources, a queue and a topic. The resource names are mapped to destination resources created in the JNDI namespace.

  @Resource(lookup = "jms/Queue")
  private static Queue queue;

  @Resource(lookup = "jms/Topic")
  private static Topic topic;

  With the common interfaces, we can mix or match connection factories and destinations. That is, in addition to using the `ConnectionFactory` interface, inject a `QueueConnectionFactory` resource and use it with a `Topic`, and can also inject a `TopicConnectionFactory` resource and use it with a `Queue`. The behavior of the application will depend on the kind of destination used and not on the kind of connection factory you use.

- **JMS Connections:** A connection encapsulates a virtual connection with a JMS provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon. We use a connection to create one or more sessions.

  Connections implement the Connection interface. When we have a ConnectionFactory object, we can use it to create a Connection:

  ```
  Connection connection = connectionFactory.createConnection();
  ```

  Before an application completes, we must close any connections created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers.

  ```
  connection.close();
  ```

- **JMS Sessions:** A session is a single-threaded context for producing and consuming messages. You use sessions to create the following:

  - Message producers
  - Message consumers
  - Messages
  - Queue browsers
  - Temporary queues and topics

  Sessions serialize the execution of message listeners. A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work.

Sessions implement the `Session` interface. After you create a `Connection` object, you use it to create a `Session`:

```
Session session = connection.createSession(false,
                           Session.AUTO_ACKNOWLEDGE);
```

The first argument means the session is not transacted; the second means the session automatically acknowledges messages when they have been received successfully.

To create a transacted session, use the following code:

```
Session session = connection.createSession(true, 0);
```

Here, the first argument means the session is transacted; the second indicates that message acknowledgment is not specified for transacted sessions.

- **JMS Message Producers:** A message producer is an object that is created by a session and used for sending messages to a destination. It implements the MessageProducer interface**.**

  We use a `Session` to create a `MessageProducer` for a destination. The following examples shows to create a producer for a `Destination` object, a `Queue` object, or a `Topic` object.

```
MessageProducer producer = session.createProducer(dest);
MessageProducer producer = session.createProducer(queue);
MessageProducer producer = session.createProducer(topic);
```

  We can create an unidentified producer by specifying `null` as the argument to `createProducer`. With an unidentified producer, we do not specify a destination until you send a message.

  Send messages by using the *`send`* method:

```
producer.send(message);
```

  If we created an unidentified producer, use an overloaded `send` method that specifies the destination as the first parameter. For example:

```
MessageProducer anon_prod = session.createProducer(null);
anon_prod.send(dest, message);
```

- **JMS Message Consumers:** A message consumer is an object that is created by a session and used for receiving messages sent to a destination. It implements the MessageConsumer interface.

  A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

  To create a `MessageConsumer` for a `Destination` object, a `Queue` object, or a `Topic` object:

```
MessageConsumer consumer = session.createConsumer(dest);
MessageConsumer consumer = session.createConsumer(queue);
MessageConsumer consumer = session.createConsumer(topic);

connection.start();
Message m = consumer.receive();
connection.start();
Message m = consumer.receive(1000); // time out after a second
```

- **JMS Message Listeners:** A message listener is an object that acts as an asynchronous event handler for messages. This object implements the MessageListener interface, which contains one method, onMessage. In the onMessage method, you define the actions to be taken when a message arrives.

- **JMS Message Selectors:** It allows a message consumer to specify the messages that interest it. Message selectors assign the work of filtering messages to the JMS provider rather than to the application.

- **JMS Messages:** The purpose of a JMS application is to produce and consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

  A JMS message can have three parts: a header, properties, and a body. Only the header is required.

  Message Headers: A JMS message header contains a number of predefined fields that contain values used by both clients and providers to identify and route messages. The below table lists the JMS message header fields and indicates how their values are set. For example, every message has a unique identifier, which is represented in the header field ***`JMSMessageID`***. The value of another header field, ***`JMSDestination`***, represents the queue or the topic to which the message is sent. Other fields include a timestamp and a priority level.

  JMS Message Header Field Values Are Set

Header Field	Set By
`JMSDestination`	`send` or `publish` method
`JMSDeliveryMode`	`send` or `publish` method
`JMSExpiration`	`send` or `publish` method
`JMSPriority`	`send` or `publish` method
`JMSMessageID`	`send` or `publish` method
`JMSTimestamp`	`send` or `publish` method
`JMSCorrelationID`	Client
`JMSReplyTo`	Client
`JMSType`	Client
`JMSRedelivered`	JMS provider

  Each header field has associated setter and getter methods, which are documented in the description of the `Message` interface. Some header fields are intended to be set by a client, but many are set automatically by the `send` or the `publish` method, which overrides any client-set values.

  Message Properties:

  We can create and set properties for messages if values are needed in addition to those provided by the header fields. We can use properties to provide compatibility with other messaging systems, or use them to create message selectors

  The JMS API provides some predefined property names that a provider can support. The use of these predefined properties or of user-defined properties is optional.

Message Bodies:

The JMS API defines five message body formats, also called message types, which allows to send and receive data in many different forms and provide compatibility with existing messaging formats. The below table describes these message types.

JMS Message Types

Message Type	Body Contains
`TextMessage`	A `java.lang.String` object (for example, the contents of an XML file).
`MapMessage`	A set of name-value pairs, with names as `String` objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
`BytesMessage`	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
`StreamMessage`	A stream of primitive values in the Java programming language, filled and read sequentially.
`ObjectMessage`	A `Serializable` object in the Java programming language.
`Message`	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

The JMS API provides methods for creating messages of each type and for filling in their contents. For example, to create and send a `TextMessage`, the following statements are used:

```
TextMessage message = session.createTextMessage();
message.setText(msg_text);     // msg_text is a String
producer.send(message);
```

At the consuming end, a message arrives as a generic `Message` object and must be cast to the appropriate message type. We can use one or more getter methods to extract the message contents. The following code fragment uses the `getText` method:

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
    TextMessage message = (TextMessage) m;
    System.out.println("Reading message: " + message.getText());
} else {
    // Handle error
}
```

JMS Queue Browsers

Messages sent to a queue remain in the queue until the message consumer for that queue consumes them. The JMS API provides a `QueueBrowser` object that allows you to browse the messages in the queue and display the header values for each message. To create a `QueueBrowser` object, use the `Session.createBrowser` method. For example:

```
QueueBrowser browser = session.createBrowser(queue);
```

The `createBrowser` method allows us to specify a message selector as a second argument when you create a `QueueBrowser`.

The JMS API provides no mechanism for browsing a topic. Messages disappear from a topic as soon as they appear: If there are no message consumers to consume them, the JMS provider removes them. Although durable subscriptions allow messages to remain on a topic while the message consumer is not active, no facility exists for examining them.

JMS EXCEPTION HANDLING

The root class for exceptions thrown by JMS API methods is `JMSException`. Catching `JMSException` provides a generic way of handling all exceptions related to the JMS API.

The `JMSException` class includes the following subclasses:

- `IllegalStateException`
- `InvalidClientIDException`
- `InvalidDestinationException`
- `InvalidSelectorException`
- `JMSSecurityException`
- `MessageEOFException`
- `MessageFormatException`
- `MessageNotReadableException`
- `MessageNotWriteableException`
- `ResourceAllocationException`
- `TransactionInProgressException`
- `TransactionRolledBackException`

J2EE CONNECTOR ARCHITECTURE

INTRODUCTION:

JCA, the J2EE Connector Architecture, is an initiative towards ***EAI, Enterprise Application Integration***. It is a standardized architecture providing the J2EE Components to have plug and play access to heterogeneous ***EIS, Enterprise Information Systems***. Examples of EIS are ***ERP (Enterprise Resource Planning), Transaction Processing Systems, Legacy Database Systems***etc.

EIS Integration:

An Enterprise Information System provides the information infrastructure for an enterprise. This information may be in the form of records in the database, business objects in an ERP, a workflow object in a ***Customer Relationship Management (CRM) System*** or a transaction program in a transaction processing application. Prior to the Internet economy, many companies had heavily invested in ***Business and Management Information Applications Systems.***

Examples of such systems can be:-

- ERP, Enterprise Resource Planning applications, such as SAP, BAAN etc.
- CRM, Customer Relationship Management applications, such as Seibel and Clarify.
- Database applications such as DB2 and Sybase.
- Main Transaction processing applications, such as CICS.
- Legacy Database Systems such as IBM's IMS.

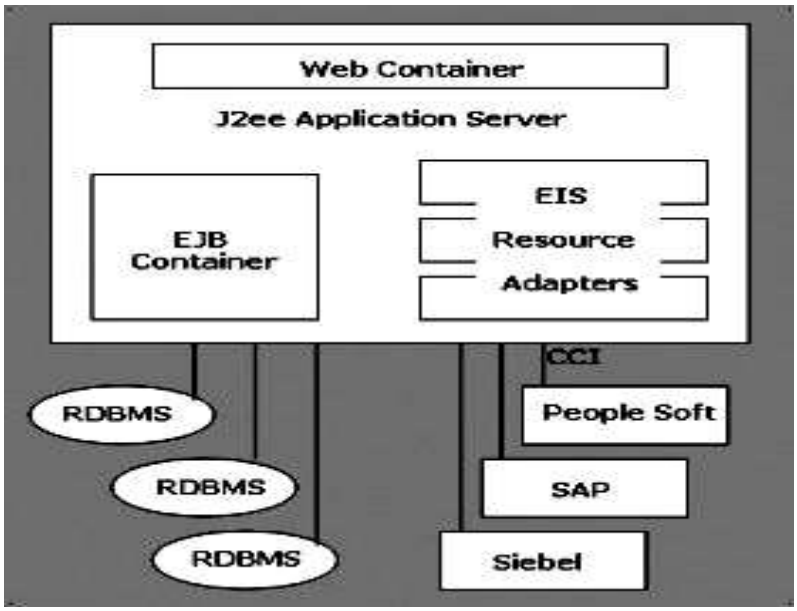
Before the JCA, most vendors supported a variety of custom adapters for integrating their EIS. Basically these adapters provided custom native interfaces, which were complex to incorporate and limited to only one

EIS. Because of this the application programmers had to deal with as many adapters as the number of EIS their application communicates to. Some of the limitations are listed below:-

1. Application programming for the EIS was proprietary in nature, because there was no generic platform for integration with the open architectures.
2. Custom adapters lacked support for ***Connection Management***, which is very crucial for large scale Web Applications, where thousands of customers interact every second. Because of which programmers had to implement connection pooling in their code which is a shear waste of time and money of the enterprises.
3. There was no standard infrastructure solution available to provide a vendor-neutral ***Security Mechanism and Generic Transaction Mechanism*** support to multiple EIS resource managers.

In order to address the above problems Sun Microsystems released the ***J2EE Connector Architecture, JCA*** that provides a standard architecture for integration of J2EE Servers with heterogeneous EIS resources. It provides a common API and a common set of services within a consistent J2EE Environment.

The following diagram shows the J2EE application server with the JCA **Components and some Enterprise Information Systems:**



In the above diagram, there is a Web Container and an EJB Container in the J2EE Application Server. There are also some blocks for Resource Adapters. Resource adapters play the role of intermediaries between the J2EE Application / Web Components and the EIS. The programmers communicate with the Resource Adapters using the ***CCI API, Common Client Interface API***. This is very similar to the JDBC API where the J2EE Components communicate with the relational database drivers using the JDBC API.

In this architecture, the J2EE Application Server implements the JCA Services like ***Connection Management, Transaction Management and Security Management*** etc. Whereas the EIS vendors implements the ***Resource Adapters*** specific to their products. This way the programmers need to code their programs in vendor neutral manner to communicate with heterogeneous EIS resources from different vendors.

Elements of the J2EE Connector Architecture

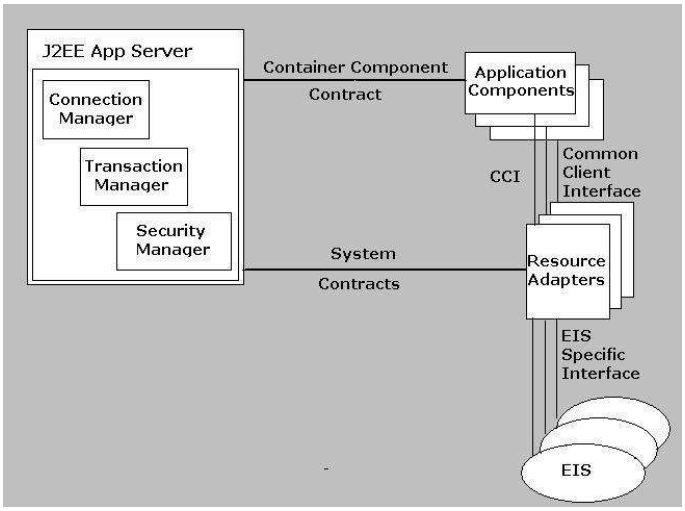
The JCA is implemented by J2EE compliant Application Servers and the Resource Adapters are provided by the EIS vendors.

The Resource Adapter is an EIS specific, pluggable J2EE Component in the Application Server, which provides an interface for the J2EE Components to communicate with the underlying EIS.

There are the following elements and Services in a JCA implementation:

1. **System-level Services**: – It defines the standard interface between the J2EE Components and the J2EE Application Server provider and the EIS Vendor. This contract specifies the roles and responsibilities of the Resource Adapter and the Application Server, so that they can co-ordinate with each other for the System level services like Connection Pooling, Security and Transactions. Furthermore this allows for any JCA compliant Resource Adapter to be pluggable to any J2EE Compliant Application Server.
2. **CCI, Common Client Interface: –** This is an API which is used by the J2EE Components and other non managed Java Applications (like standalone Java programs and Java Applets) to communicate with the Resource Adapters, which in turn communicate to the underlying EIS.
3. **Packaging and Deployment Interfaces: –** Packaging and Deployment Interfaces allow the Resource Adapters to be able to plug into any Application Server.

The following diagram shows all the components of the J2EE Connector Architecture. It shows how the components communicate with each other.



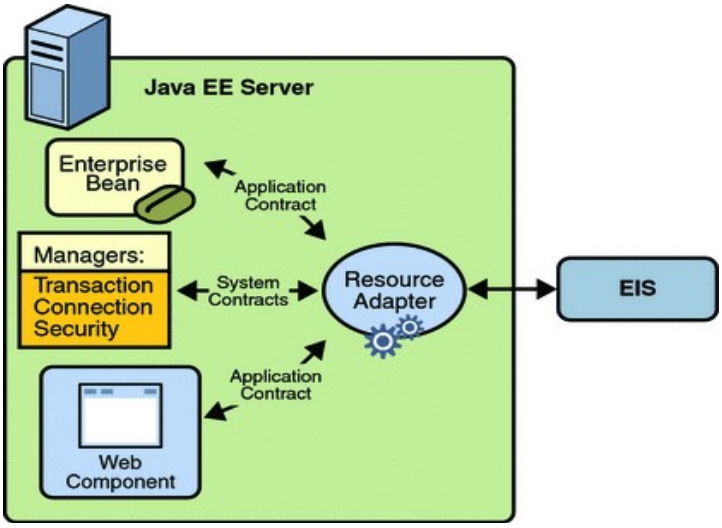
The CCI, Common Client Interface is part of the implementation of the Resource Adapters. The J2EE components use this API in order to communicate with the underlying EIS resource. Because of the System Level contracts the J2EE Application Servers provide the services like Connection Pooling, Transaction Management and Security Management and the programmers can stay away from the implementation of these services.

There are two types of environments based on the type of client application using the Resource Adapter. These are:

- **Managed Environment:** This includes the multi tired J2EE based Web Applications that may have several Web / Application Components communicating to the Resource Adapters. Such applications are called as the Managed Applications in the JCA context.
- **Non-Managed Environment:** In a two tired architecture, where the application client directly uses the application, the Resource Adapter provides the system level services to its clients. Such applications are referred as Non Managed applications.

RESOURCE ADAPTER

The Resource Adapter is a component in the J2EE Connector Architecture that sits between the J2EE Components and the EIS. This module is implemented with the EIS specific library, which can be written in Java or with native interface components. Basically it implements two things, first is the Common Client Interface, CCI so that when the developers invoke the methods of this library in their code, the invocations can be resolved by the implementations inside the Resource Adapter, and second is the implementation of the functionality through which it connects to the underlying EIS resource and get the work done for the method invocations. The Resource Adapters are required to support the System Level contracts as is required by the Application Servers. There are mainly two types of contracts that a Resource Adapter implements in order to get compliant with the JCA. These are:



- **Application-Level Contracts:** Application Contract basically defines the CCI that the Resource Adapter must implement in order for the J2EE Components and the non-managed components to communicate to the underlying EIS resource.
- **System-Level Contracts: –** This defines a set of System Contracts which enable the Resource Adapter to plug-in into the Application Server and utilize its services to manage connections, transactions and Security. The three basic contracts are as follows:
  - **Connection Management:** The Connection Management contract specifies the following things:
    1. Connection to the Resource Adapters for managed and non-managed components can be made by using consistent application programming techniques, which remain same every time.
    2. The Resource Adapter provides the Connection Factory and Connection Interface based on the CCI, which in turn is specific to the EIS.
    3. Provide a generic mechanism for the J2EE components to consume the Application Server services like Transaction, Security and last but not the least the Connection Pooling etc.
  - **Transaction Management:** This contract extends the Application Server's transactional capabilities to the underlying EIS Resource Managers. An EIS Resource Manager manages a set of shared EIS resources to participate in a transaction. It can manage the following two types of transactions: –
    1. **XA Transaction:** XA Transactions are controlled and coordinated by external Transaction Managers. A JTA XATransaction Management contract exists between the JCA compliant Resource Adapters and its underlying Resource Manager. The participating EIS resource also supports XA Transactions by implementing an XAResource in their Resource Adapter. The JTA XAResource interface enables two resource managers to participate in transactions coordinated by an external Transaction Manager. This allows the transactions to be managed by a transaction manager which is external to the Resource Adapter.
    2. **Local Transactions: –** These transactions do not require any external Transaction Manager, because it is managed internally, either by the J2EE Application Server (container managed) or by the J2EE component (component managed). In Component Managed Transactions the component uses JTA UserTransaction interface or a transaction API specific to the EIS. When an application component requires an EIS Connection, the Application Server starts a local transaction using the currently available transaction context. Upon closure of the connection by the application component, the server commits or rolls back depending upon success or failure of the transaction.
  - **Security Management:** This contract defines ways to implement security between the Application Server and the EIS resource. There are the following mechanisms used to protect the EIS against unauthorized access and other security threats:
    1. Use of user identification, authentication and authorization.

2. Use of open network communication security protocols like Kerberos which provide end to end security with authentication and confidentiality services.
3. Implementing an EIS specific security Service.

EIS Sign-On

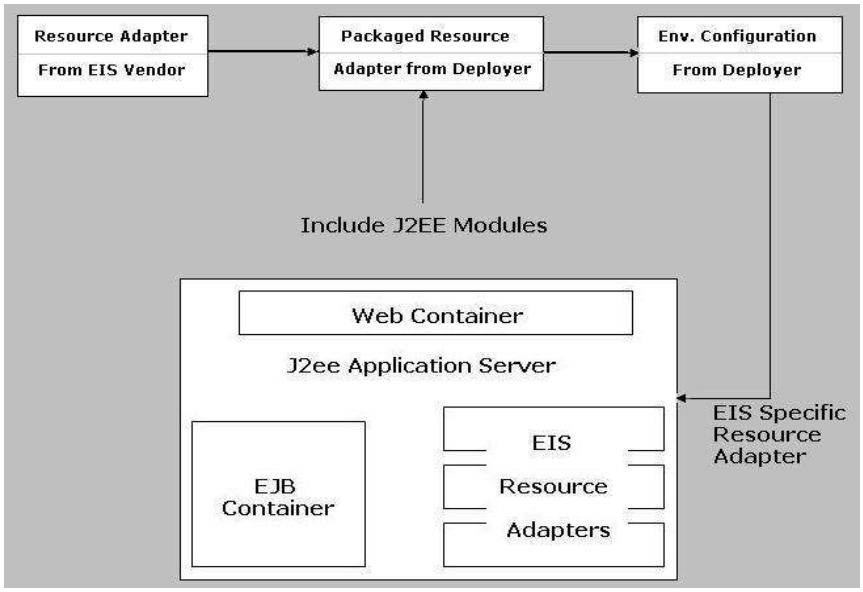
A Sign-on process is required while creating a new connection to EIS. Based on the security context, authentication and authorization is done for the user to obtain appropriate access. A re-authentication is required in case the Security Context is changed for the user. The following steps are required for sign on:

1. Identity of the initiating user, or ***Resource Principal***, is determined. This helps finding the security context of the user using which the connection will be established to the EIS.
2. Resource Principal is authenticated if the connection is not already authenticated.
3. Establish a secure connection between the Application Server and the EIS Resource using secure mechanisms like SSL or Kerberos etc. At this point the connection is associated with the initiating user. All subsequent invocations will happen under this context.

This sign-on process can be Application Managed or it can be Container Managed. In case of Application Managed sign-on process, the application component itself provides the security credentials to the EIS Resource Adapter. While in the later case it becomes the duty of the application to find out the Resource Principal of the initiating user and present information of this Resource Principal to the EIS Resource Adapter in the form specific to the ***JAAS, Java Authentication and Authorization Service***.

Packaging and Deploying a Resource Adapter

The Resource Adapters module is packaged as a .rar file, similar to the .ear files for the J2EE Application Modules. There are proper deployment and packaging interfaces that allow a Resource Adapter to plug-in to any J2EE compliant Application Server. The following diagram illustrates the process of packaging and deployment of Resource Adapters module into the Application Server.



The exact steps involved in packaging and deployment are listed below:

1. The EIS Resource Adapter provider, usually the EIS Vendor, implements the Resource Adapter as a set of Interfaces and utility classes implementing the CCI API and the EIS functionality. These implementations take care of the JCA contracts that the Resource Adapter need to follow for JCA compliance.
2. All the components of the Resource Adapter, including the source files are packaged into a .rar file, called as ***Resource Adapter Module***. The deployment descriptor of the Resource Adapter specifies the contract between the Resource Adapter provider and the one who deploys the Resource Adapter.
3. In the actual deployment process the Resource Adapter is deployed in the Application Server and then it is configured with the Application Server and the underlying EIS environment.

Packaging a Resource Adapter

The Resource Adapters are proper J2EE components packaged in .rar files. One or more Resource Adapters can be staged in one directory and then packaged as .rar files. The following steps are usually involved for packaging a Resource Adapter:-

1. Compile the Resource Adapter Java files into a staging directory.
2. Create a .jar file to add the class files created in the above step.
3. Create an ra.xml deployment descriptor file in the META-INF subdirectory. Add entries for the Resource Adapter.
4. Create a J2EE Application Server specific deployment descriptor in the same directory and add the entries for the Resource Adapter.
5. Create the Resource Adapter module .rar file by executing the following command:

```
1 jar cvf JavaBeatResourceAdapter.rar -C staging-dir
```

Now this .rar file can be deployed to the Application Server or packaged inside an application's .jar file. The packaged Resource Adapter includes the following:

1. Java Classes and Interfaces that implement the functionality of the JCA contract and the underlying EIS resource.
2. Some helper classes used by main implementation classes.
3. Platform dependent native libraries required by the Resource Adapter.
4. Meta information about the above elements.
5. Documentation.

After installation of the .rar file, the directory structure of the Application Server looks like the following: –

1 \AppServerHomeDir

2 \config

3 \JavaBeatDomain

4 \applications

5 \JavaBeatResourceAdapter.rar

There are basically two deployment descriptors packaged within a .rar file of the ***Resource Adapter Module***. These are the ra.xml, which specifies the general information about the Resource Adapter, and the ABCAppServer.xml, which specifies operational parameters required for the Application Server.

Ra.xml, the Resource Adapter deployment descriptor

Following is a deployment descriptor for an ABC Resource Adapter.

<connector>

<display-name>ABCResourceAdapter<display-name>

<vendor-name>Java Beat</vendor-name>

<spec-version>1.0</spec-version>

<eis-type>JDBC Database</eis-type>

<version>1.5</version>

<resourceadapter>

<managedconnectionfactory -class>
com.sun.connector.javabeat.NoTxManagedConnectionFactory</managedconnectionfactory-class>

<connectionfactory-interface>javax.sql.Datasource</connectionfactory-interface>

13 <connectionfactory-impl-class> com.sun.Connector.abc.JdbcDataSource</connectionfactory-impl-class>

14 <connection-interface>java.sql.Connection</connection-interface>

15 <connection-impl-class>com.sun.connector.abc.JdbcConnection</connection-impl-class>

16 <transaction-support>NoTransaction</transaction-support>

```
<config-property>
<config-property-name>ConnectionURL</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value>jdbc:cloudspace:rmi:CloudscapeDB:create=true</config-property-value>
</config-property>
<authentication-mechanism>
<authentication-mechanism-type>BasicPassword</authentication-mechanism-type>
<credential-interface>javax.resource.security.PasswordCredential</credential-interface>
</authentication-mechanism>
<reauthentication-support>false</reauthentication-support>
</resourceadapter>
</connector>
```

The first block of the descriptor is:-

```
<display-name>ABCResourceAdapter<display-name>
<vendor-name>Java Beat</vendor-name>
<spec-version>1.0</spec-version>
<eis-type>JDBC Database</eis-type>
<version>1.5</version>
```

It lists general information about the Resource Adapter, like its name, Vendor's name, specification version, the type of the EIS resource that this Adapter is representing and the version of the JCA supported.

Then the name of the class that implements the javax.resource.spi.ManagedConnectionFactory interface is listed in the element.

The ConnectionFactory interface and the implementation class are mentioned thereafter in the following block of the descriptor:-

```
<connectionfactory-interface>javax.sql.Datasource</connectionfactory-interface>
<connectionfactory-impl-class>
com.sun.Connector.abc.JdbcDataSource</connectionfactory-impl-class>
```

The Connection implementation class for the java.sql.Connection interface is com.sun.connector.abc.JdbcConnection.

The level of transaction support can be either *NoTransaction*, or *LocalTransaction* or*XATransaction*. In this example the Resource Adapter does not support transaction which is specified by the *NoTransaction* value in the element.

Certain properties of *ManagedConnectionFactory* can be configured by specifying a name and type of the property and then providing a value of the property as follows: –

```
<config-property>
<config-property-name>ConnectionURL</config-property-name>
<config-property-type>java.lang.String</ config-property-type>
<config-property-value>jdbc:cloudspace:rmi:CloudscapeDB:create=true</ config-property-value>
</config-property>
```

The last section of the deployment descriptor specifies the authentication mechanisms supported by the Resource Adapter provider. In this case only *BasicPassword* method is supported. The Security Credential interface is *javax.resource.security.PasswordCredential*.
There is no re-authentication support with this Resource Adapter.

```
<authentication-mechanism>
<authentication-mechanism-type>BasicPassword</authentication-mechanism-type>
<credential-interface>javax.resource.security.PasswordCredential</credential-interface>
</authentication-mechanism>
<reauthentication-support>false</reauthentication-support>
```

The Resource Adapter can be deployed in two ways as discussed below: –

1. By using the deploytool command line or the deploytool UI console using the following command: –

```
Deploytool -deployConnector %J2EE_HOME%\lib\connector\abc-ra.rar <HostName>
```

2. Deploying as a Web Application archive file, i.e. .ear file. In this method the .rar file is included into the .ear file just as we include a .jar or .war file in it. And then the .ear file is deployed in the Application Server. To identify the .rar module in the package the following line is added into the application.xml file: –

```
<connector>abcresourceadapter.rar</connector>
```

JCA Advantages

J2EE Connector Architecture has emerged as the primary technology for providing a vendor neutral platform for the programmers to communicate with multiple EIS resources without changing their components at all. There are the following scenarios in which the technology offers a potential solution:

1. **Enterprise Application Integration:** While integrating multiple EIS applications there are many challenges that need to be taken care of. These include, transaction, security and scalability issues. Multiple EIS vendors providing their own proprietary APIs for integration, which creates much bigger problem of embedding code specific to each vendor into the application components. But with the JCA technology, the EIS vendors are now required to provide with a Resource Adapters obeying the JCA contracts. And the programmers need to understand just the CCI API, required to communicate with the similar Resource Adapter of all the different EIS resources.
2. **Web-Enabled Enterprise Portals:** Enterprise portals usually unifies all the information, services, applications and processes for all the members of an enterprise, including the employees, clients, partners and customers etc. Such a portal needs to integrate all the heterogeneous EIS resources, such as database systems, information systems etc. which handle all information of the enterprise. In such a scenario the JCA technology solves the problem by seamlessly integrating the different EIS resources by introducing the concept of Resource Adapters and the CCI API. It also helps by providing the EIS sign-on mechanism that supports generic and application specific authentication mechanism.
3. **Business to Business Integration:** Business to Business integration requires end-to-end process automation and allows application interaction with an enterprise and across partners through the internet. In order to support the external partner interactions, the backend internal business systems need to be seamlessly integrated into the same process. With the help of JCA the backend EIS resources and end to end processes can be integrated without much difficulty.