LIYANA ARACHCHIGE RANIL

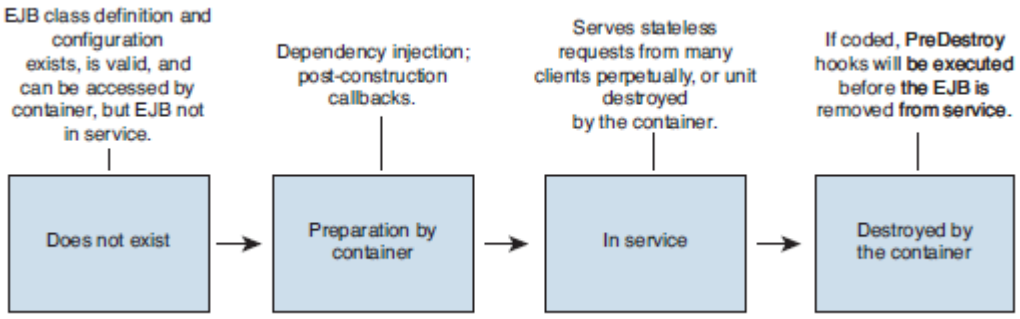
SHORT NOTES / MARK CADE / HUMPHY SHEIL / BUSINESS TIER TECHNOLOGIES

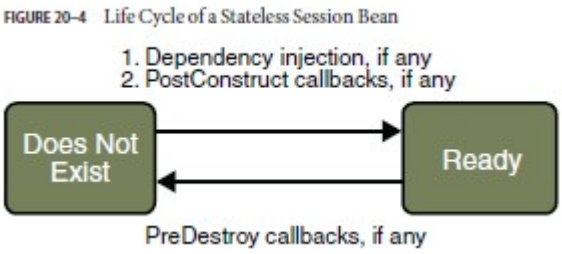
1. **EJB** is a **SERVER SIDE COMPONENT** used in J2EE architecture to **ENCAPSULATE** a **SPECIFIC PIECE** of **BUSINESS LOGIC**
2. **EJB(S)** are **DISTRIBUTED** COMPONENTS
3. They can be **ACCESSED LOCALLY** or **REMOTELY**
4. **EJB(S) SIMPLIFIES** the development of **LARGE** , **DISTRIBUTED** application , since EJB container provides **SYSTEM-LEVEL SERVICES** to EJB(S), then the bean **DEVELOPER** can **FOCUS** on **SOLVING** the **BUSINESS PROBLEM**
5. **BEAN** contains the **BUSINESS LOGIC**, not the **CLIENT**. Hence **CLIENT DEVELOPER** can focus on **PRESENTATION** of the **CLIENT**. **As a result CLIENT**S are **THINNER**
6. **EJB(S)** are **MANAGED** by the **CONTAINER** , with container providing important services such as **TRANSACTION MANAGEMENT** , **SECURITY , CONCURRENCY CONTROL**
7. **WHEN TO USE EJB**
    a. **–** For the applications that **MUST be SCALABLE**
    b. **– TRANSACTION MUST ENSURE DATA INTERGRITY**
    c. **–** The **Application** will have **VARIETY** of **CLIENTS**
8. **EJB(S) DO NOT** manage **INBOUND** or **OUTBOUND DATA** , the CONTAINER manages **ALL CLIENT ACCESS**
9. If **WRITTEN** using **ONLY** those services **DEFINED** by the **SPECIFICATION , EJB(S)** can be **PORTED** to other **EJB CONTAINERS** with **MINIMAL EFFORT**
10. **EJB** has few major releases , **EJB 2.0** , **EJB 2.1** and **EJB 3.0**
11. **EJB 3.0** was completely targeted for **EASE of DEVELOPMENT**

12. **EJB 3.0** specification has THREE parts , ejb-core , ejb-simplified , jpa
13. Overall **EJB COMPONENT** model **ALLOWS** the following **HIGH LEVEL** characteristic
    a. A **STATELESS SERVICE** , including the **ABILITY** to act as **a WEB SERVICE END POINT**
    b. A **STATEFUL SERVICE**
    c. A service invoked **ASYNCHRONOUSLY** by a **SEPARATE COMPONENT**
    d. An **ENTITY OBJECT**

SESSION BEANS – STATELESS/STATEFULL [EJB 3.0]

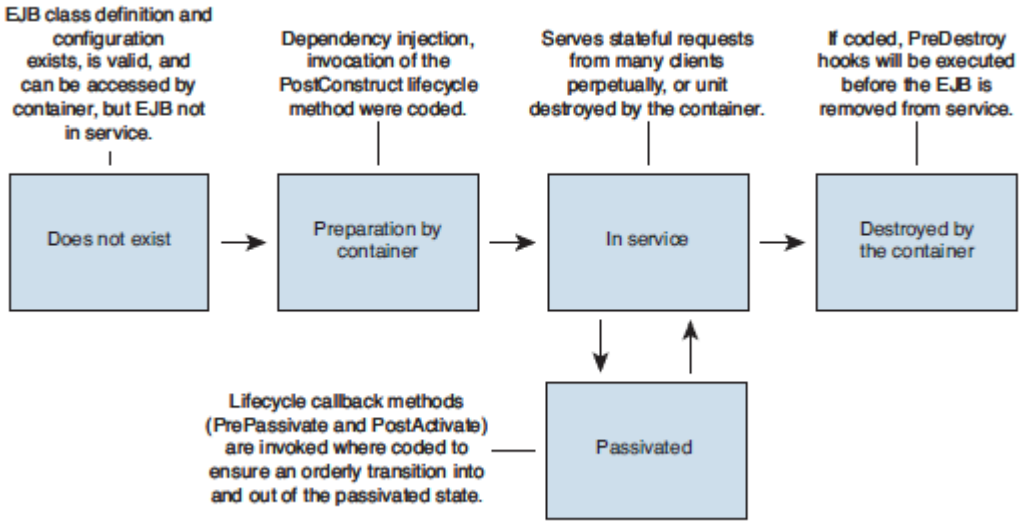
1. **SESSION BEANS** are EJB(S) that CONTAIN BUSINESS LOGIC
2. **SESSION BEANS** would contain **BUSINESS LOGIC** that is related to the **IMPLEMENTATION** of a **WORKFLOW** or **PROCESS**
3. **SESSION BEANS** can be considered as **SERVER SIDE PROXIES**
4. **SESSION BEANS** are **NOT PERSISTANT**
5. **TWO MAJOR TYPE OF SESSION BEAN – STATELESS SESSION BEANS , STATEFUL SESSION BEANS**
6. **STATELESS SESSION BEANS do not** maintain **any INTERNAL CLIENT** SPECIFIC state across SEPARATE CLIENT invocations
7. **STATELESS SESSION BEAN** does **NOT** MAINATIN a **CONVERSATIONAL** state with the **CLIENT**
8. **ALL** instances of **STATELESS BEANS** are **EQUIVALENT**
9. A **STATELESS SESSION BEAN** can **IMPLEMENT** a **WEB SERVICE**, Other types of EJB **CAN NOT**
10. Application SCALABILITY is IMPROVED GNIFICANTLY when using STATELESS SESSION beans

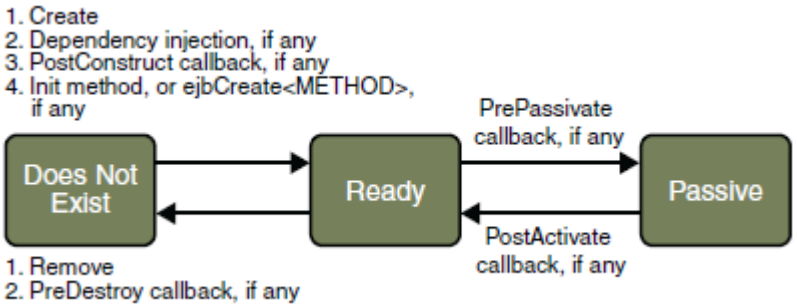
11. A SMALL POOL of (relative to the SIZE of the number of CONCURRENT request) STATELESS SESSION BEANS can be used to service a SIGNIFICANTLY larger number of CONCURRENT REQUESTS
12. **STATELSS SESSION BEANS** register **EJB TIME SERVICE** to receive **EVENT NOTIFICATIONS** by adding **@Timeout** annotation
13. **STATELESS SESSION BEAN life cycle**
14. 

15. **STATELESS SESSION BEANS** are **NEVER** PASSIVATED
16. 

17. **TIMER SERVICE** can be **USED** by **ALL TYPES OF EJB(S) EXCEPT STATEFUL SESSION BEANS**

18. **STATEFUL ENTERPRISE BEANS** has the **ABILITY** to **MAINTAIN INTERNAL CONVERSATIONAL STATE** across **MULTIPLE INVOCATIONS** from the **SAME CLIENT**
19. Applications that **use STATEFUL SESSION BEANS** are **NOT** as **scalable** as **EQUIVALENT** application that uses **STATELESS SESSION** BEANS
20. STATEFUL BEANS can be ACTIVATED and PASSIVATED by the CONTAINER
21. **STATEFUL SESSION BEANS** can not be REGISTERED with EJB TIMER SERRVICE
22. **STATEFULL SESSION BEANS life cycle**
23.





24.

25. Whenever the STATEFUL EJB is created the CONTAINER PERFORMS
    e. DEPENDENCY INJECTION
    f. POST CONTRUCT CALL BACKS if ANY
26. **EJB CONTAINER** would **INVOKE** @**PrePassivate** method if ANY **BEFORE PASSIVATING** STATEFUL BEAN
27. When EJB **CONTAINER ACTIVATES** A **STATEFUL** BEAN it would **INVOKE @PostActivate** method if ANY
28. At the **END** of the LIFE CYCLE , the **CLIENT INVOKES** a **METHOD** annotated with **@Remove**
29. And the EJB **CONTAINER CALLS** the **METHOD** annotated **@PreDestroy** if ANY
30. Your CODE controls the INVOCATION of ONLY ONE LIFE CYCLE method, the method annotated with **@Remove**
31. STATEFUL SESSION BEANS are appropriate for
    g. The BEANS state REPRESENTS the INTERACTION between the BEAN and a SPECIFIC client
    h. The BEAN needs to host INFORMATION about the CLIENT ACROSS method INVOCATIONS
32. **ALL SESSION BEANS** require SESSION BEAN class
33. **SESSION BEANS** can implement MORE THAN ONE INTERFACES
34. **TYPES** of **CLIENTS** can be **REMOTE** , **LOCAL** or **WEB SERVICE**

35. **REMOTE CLIENTS – runs on different JVM , can be web component , an application client or another EJB, location of the EJB is TRANSPARENT**
36. **LOCAL CLIENT –** MUST run on the SAME JVM as the EJB ,can be a WEB COMPONENT or another EJB, the LOCATION of the EJB is not TRANSPARENT
37. The **SAME** business interface CAN NOT BE both LOCAL and REMOTE
38. **DECIDING** on **LOCAL** or **REMOTE** access
    i. **Tighter or Loose coupling** (tightly coupled beans are good candidates for LOCAL access. They typically call each other often , this would enhance PERFORMANCE)
    j. **Type of Client –** If an Application Client ,then Remote . If the clients are web components or other EJB, then type of access DEPENDS on HOW YOU DISTRIBUTE your COMPONENTS
    k. **Component Distribution –**
    l. **Performance –** Network latency would degrade the Performance (Remote calls may be slower than the Local calls) while if the component are distributed , those can be deployed in different servers and improve overall performance
39. All EJBs that **PERMIT REMOTE** access **MUST** have a **REMOTE** business **INTERFACE**
40. The **BUSINESS INTERFACE** is **LOCAL** interface **UNLESS** it is annotated with **@Remote**
41. If **the BEAN CLASS** implements a **SINGLE INTERFACE** , the **INTERFACE** is **ASSUMED** to be the **BUSINESS INTERFACE**

42. If the **BEAN CLASS** implements **MORE THAN ONE** interface , either the **BUSINESS INTERFACE** must be **EXPLICITLY** annotated either **@Local** , **@ Remote** or the **BUSINESS INTERFACE** must be **SPECIFIED** by **DECORATING** the **BEAN CLASS** with **@Remote or @Local**
43. **java.io.Serializable , java.io.Externalizable** , any interface **DEFINED** in java.exj package are **IGNORED** in the above case
44.

```
package com.sun.tutorial.javaee.ejb;

import java.util.List;
import javax.ejb.Remote;

@Remote
public interface Cart {
    public void initialize(String person) throws BookException;
    public void initialize(String person, String id)
         throws BookException;
    public void addBook(String title);
    public void removeBook(String title) throws BookException;
    public List<String> getContents();
    public void remove();
}
```

45.

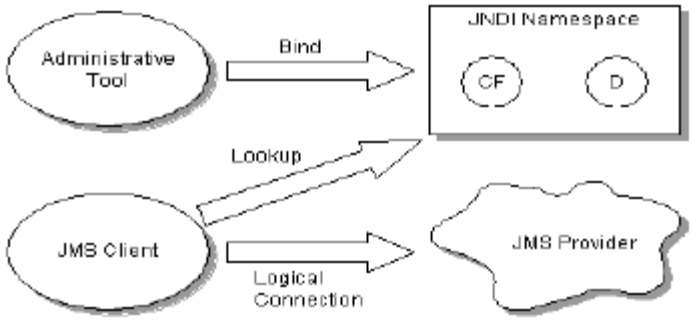
46. **@postConstruct** methods are invoked by the **CONTAINER** on **NEWLY** constructed bean instances **AFTER** all **DEPENDENCY INJECTIONS** has completed and **BEFORE the FIRST BUSINESS METHOD** is invoked
47. **@PreDestroy** methods are invoked **AFTER** any **METHOD** annotated **@Remove** has **COMPLETED** and **BEFORE** the **CONTAINER REMOVES** the EJB Instance
48. **@PostActivate** methods are invoked by the CONTAINER after the CONTAINER MOVES The BEAN from SECONDARY STORAGE to ACTIVE STATUS
49. **@PrePassivate** methods are INVOKED by the CONTAINER BEFORE the CONTAINER PASSIVATES the EJB
50. The SINGNATURE of a BUSINESS METHOD MUST conform to following RULES
    m. METHOD name MUST not BEGIN with ejb to AVOID CONFLITS with callback methods defined by EJB architecture
    n. The Access control modifier MUST be PUBLIC
    o. If the BEAN allows remote access , the ARGUMENTS and RETURN types MUST be LEGAL types for JAVA RMI
    p. If the BEAN class is capable of being a WEB SERVICE END POINT the methods exposed as WEB service METHODS must have arguments and return types compatible to JAXB
51. **STATEFULL SESSION BEANS** have **@Remove** method declared usually. The container REMOVES the bean after @Remove method completes

MESSAGE DRIVEN BEANS (MDB)

1. **MESSAGE DIRVEN BEANS (MDB)** receive and process messages from a **JMS** destination
2. **JMS Destination** can be a **QUEUE** or **TOPIC**
3. **JMS** Specification defines a common way for JAVA programs to create , send , receive and read an ENTERPRISE MESSAGING SYSTEM'S messages
4. **Enterprise Messaging Products** are sometimes called Message Oriented Middleware (MOM)
5. **JMS** is a set of interfaces and associated semantics that define how a JMS client accesses the facilities for an ENTERPRISE MESSAGING PRODUCT
6. **Messaging is** peer-to-peer
7. **JMS Provider** is the ENTITY that implements JMS for a MESSAGING PRODUCT
8. **A** major goal of JMS is that CLIENTS have a CONSISTANCT API for creating and working with MESSAGES that is INDEPENDENT fof the JMS PROVIDER
9. **Messaging products** can be POINT-TO-POINT or PUBLISH-SUBSCRIBE systems
10. **POINT-TO-POINT** = Built around the concept of MESSAGE **QUEUE**(S). Each message is addressed to a SPECIFIC QUEUE. Clients EXTRACT messages from the QUEUE(S) established to HOLD their MESSAGES (PQ, pointtopoint=queue)
11. **PUBLISH and SUBSCRIBE (PUB/SUB)** clients address messages to some NODE in a content hierarchy. PUBLISHERS and SUBSRIBERS are generally ANONYMOUSE
12. The system takes care of DISTRIBUTING the messages arriving from a NODE'(S) multiple PUBLISHERS to its MULTIPLE SUBSCRIBERS

13. **JMS** does not provide
    a. Load balancing / fault tolerance
    b. Error / Advisory notifications
    c. Administration of messaging products
    d. Security – **JMS** does **NOT** specify an API for controlling **PRIVACY** and **INTERGRITY**. It is expected that many KMS providers will provide such features. It is also expected that configuration of these services will be handled by **PROVIDER-SPECIFIC** way
    e. Wire protocol
    f. Message type repository
14. **MDB(S)** provide ability to ASYNCHRONOUSLY process messages
15. **EJB** supports both **SYNCHRONOUS** and **ASYNCHRONOUS** message consumption. The synchronous one is via METHOD calls and ASYNCHRONOUS one using BEANS which is invoked when JMS client sends it a message (MDB)
16. **JMS 1.1** introduced a unified interfaces to interact with the messages from both DOMAINS , point-to-point and publish/subscribe
17. **PRIOR** to **JMS 1.1** , the there were **TWO DIFFERENT** sets of **INTERFACES** for both **DOMAINS** , namely , **PUB/SUB** and **POINT-to-POINT**
18. A JMS application has ,
    a. **JMS Client** – Java language programs that sends and receives messages
    b. **Non-JMS Client** – Clients that uses MESSAGING SYSTEMS NATIVE client API instead of JMS. If the application predated the availability of JMS it is likely that it will include both JMS and NON-JMS clients

c. **Messages** – Each application defines set of messages that are used to communicate information between clients
d. **JMS Provider** – Messaging System that IMPLEMENTS JMS in addition to other functionality
e. **Administered Objects** – Administered objects are PRE-CONFIGURED JMS objects created by an ADMINISTRATORY for use by clients. Administered objects are usually placed in JNDI namespace by ADMINISTRATOR
    i. **ConnectionFactory** – Uses to create a Connection with the provider
    ii. **Destination** – uses to specify the destination of messages



f.
g. Administered objects are placed in JNDI namespace by an Administrator

19. An application can **COMBINE** both styles of domain , "**Point-to-Point**" , "**Pub/Sub**"

20.

JMS Common Interfaces	PTP-specific Interfaces	Pub/Sub-specific interfaces
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

21. **JMS DOES NOT** provide features for controlling or configuring **MESSAGE INTERGRITY** or **MESSAGE PRIVACY**
22. **JMS** provides the **JMSReplyTo** message header field for specifying the Destination where a reply to a message should be sent
23. **JMSCorrelationID** header field of the REPLY can be used to **REFERENCE** the **ORIGINAL REQUEST**
24. **JMS** message consist of **HEADER** , **PROPERTIES** and **BODY**
25. **Message Header Fields : JMSDestination** [destination to which the message is sent] , **JMSDeliveryMode** [delivery mode specified when the message was sent],**JMSMessageID**[value that uniquely identifies each message sent by the provider],**JMSTimestamp**[time a message was handed off to a provider to be sent],**JMSCorrelationID**[link one message with another],**JMSReplyTo**[destination supplied by the CLIENT when a message is sent,destination where to reply should be sent],**JMSRedelivered**[it is likely , but not guaranteed that this message was delivered but not acknowledged in the past],**JMSType,JMSExpiration,JMSPriority**

Table 3-1 Message Header Field Value Sent

Header Fields	Set By
JMSDestination	Send Method
JMSDeliveryMode	Send Method
JMSExpiration	Send Method
JMSPriority	Send Method
JMSMessageID	Send Method
JMSTimestamp	Send Method
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	Provider

26.

27. **JMS** message body could be **StreamMessage**[Stream of JAVA primitive values] **, MapMessage** [set of Name-Value pairs]**, TextMessage , ObjectMessage**[Contains serializable JAVA object] **, BytesMessage** [Stream of Un interpreted bytes]
28. A **JMS SESSION** may optionally be **TRANSACTED**
29. Each **TRANSACTIONAL SESSION** supports **SINGLE SERIES of TRANSACTIONS**
30. Each **TRANSACTION GROUPS** a set of **PRODUCED MESSAGES** and a **SET OF CONSUMED** messages into an **ATOMIC** unit of work
31. When a **TRANSACTION commits**, its **ATOMIC unit of INPUT is ACKNOWLEDGED** and **its ASSOCIATED ATOMIC UNIT of OUTPUT is SENT**. If **ROLLBACK** is done ,its **PRODUCED MESSAGES** are **DESTROYED** and its **CONSUMED** messages are automatically **RECOVERED**
32. **Distributed Transactions** are **NOT** required to be **SUPPORTED** by **JMS** , but if it does so then it **MUST** use **JTA XAResource API**
33. **JMS** defines that **MESSAGES sent by a particular SESSION to a destination MUST BE RECEIVED in the order in which they were SENT**
34. **JMS** does **NOT** define **ORDER** of **MESSAGE RECEIPT ACROSS destinations** , or **across DESTINATION MESSAGES sent from MULTIPLE SESSIONS**
35. If a **SESSION** is **TRANSACTED** , message **ACKNOWLEDGEMENT** is handled **automatically by COMMIT** and **recovery is handled automatically by ROLLBACK**
36. If **SESSION is NOT TRANSACTED** , there are **THREE ACKNOWLEDGEMENT** options and **RECOVERY is handled MANUALLY**

a. **DUPS_OK_ACKNOWLEDGE**
    i. Session would **LAZYLY ACKNOWLEDGE** the delivery of messages. Would **RESULT** in delivery of **DUPLICATE** messages
b. **AUTO_ACKNOWLEDGE**
    i. Session **AUTOMATICALLY** acknowledges a client's receipt of a message when it has successfully returned from a call to **RECEIVE**
c. **CLIENT_ACKNOWLEDGE**
    i. Client acknowledges a message by calling message's **acknowledge** method

37. A client uses a MessageConsumer to receive messages from a Destination.
38. A **MessageConsumenr** is created by passing **QUEUE** or **TOPIC** to a **SESSION'S createConsumer** Method
39. A Consumer can be created with a **MESSAGE SELECTOR**. This allows the client to RESTRICT the messages delivered to the **CONSUMER**
40. A Client MAY either **SYNCHRONOUSLY** receive consumer's messages or have the **PROVIDER ASYNCHRONOUSLY** deliver them as they arrive
41. **SYNCHRONOUS MESSAGE DELIVERY** : A client can request next message from a **MESSAGE CONSUMER** using one of its **receive** methods

a.

To receive the next message in the *Queue*, you can use the *MessageConsumer.receive* method. This call blocks indefinitely until a message arrives on the *Queue*. The same method can be used to receive from a *Topic*.

```
TextMessage stockMessage;
stockMessage = (TextMessage)receiver.receive();
```

To limit the amount of time that the client blocks, use a timeout parameter with the *receive* method. If no messages arrive by the end of the timeout, then the *receive* method returns. The timeout parameter is expressed in milliseconds.

```
TextMessage stockMessage;

/* Wait 4 seconds for a message */
TextMessage = (TextMessage)receiver.receive(4000);
```

42. **ASYNCHRONOUS MESSAGE DELIVERY**: A client can register an object that implements the **JMS MessageListener** interface with **a MESSAGE CONSUMER**. As messages arrive for the **CONSUMER**, the provider delivers them by **CALLING** the listeners **onMessage**() method

a.

MessageListener interface, called *StockListener.java*, might look like

```
import javax.jms.*;

public class StockListener implements MessageListener
{

    public void onMessage(Message message) {
        /* Unpack and handle the messages received */
        ...
    }
 }
```

The client program registers the *MessageListener* object with the *MessageConsumer* object in the following way:

```
StockListener myListener = new StockListener();

/* Receiver is MessageConsumer object */
receiver.setMessageListener(myListener);
```

43. There are **TWO JMS delivery MODES**
    a. **NON_PERSISTENT** [at-most-once , messages can get lost / performance is high]
        i. Lowes overhead delivery mode. Does not require that the message be logged to stable storage. A JMS provider failure can cause a NON_PERSISTENT message to be lost

b. **PERSISTENT** [once –and-only-once , must not get lost , must not duplicate either / reliability is high]
    i. Instructs the JMS provider to take EXTRA care to INSURE the message is not lost in transit due to a JMS provider failure

44. JMS supports following administered objects a multithreading access

designed to be accessed by one logical thread of co

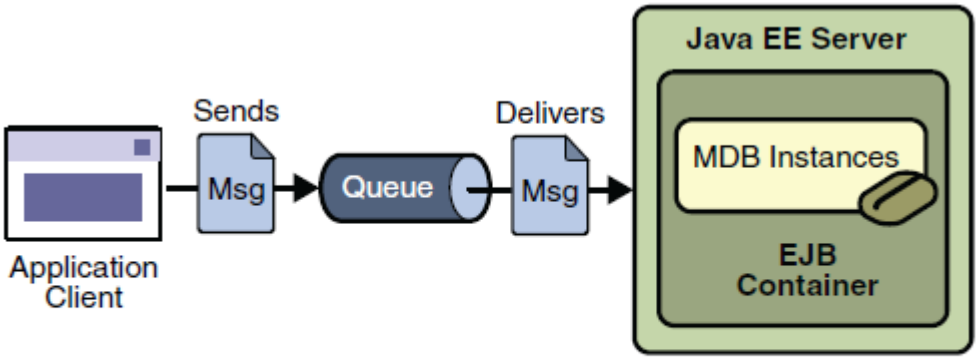
Table 2-2 JMS Objects that Support Concurrent Use

JMS Object	Supports Concurrent Use
Destination	YES
ConnectionFactory	YES
Connection	YES
Session	NO
MessageProducer	NO
MessageConsumer	NO

a.

45. **MDB(S)** are probably the SIMPLEST type of EJB because there is only ONE METHOD [**onMessage**]
46. **MDB(S)** can be used to **COMSUME MESSAGES** from any **CONNECTOR 1.5 RESOURCE ADAPTOR**

47.



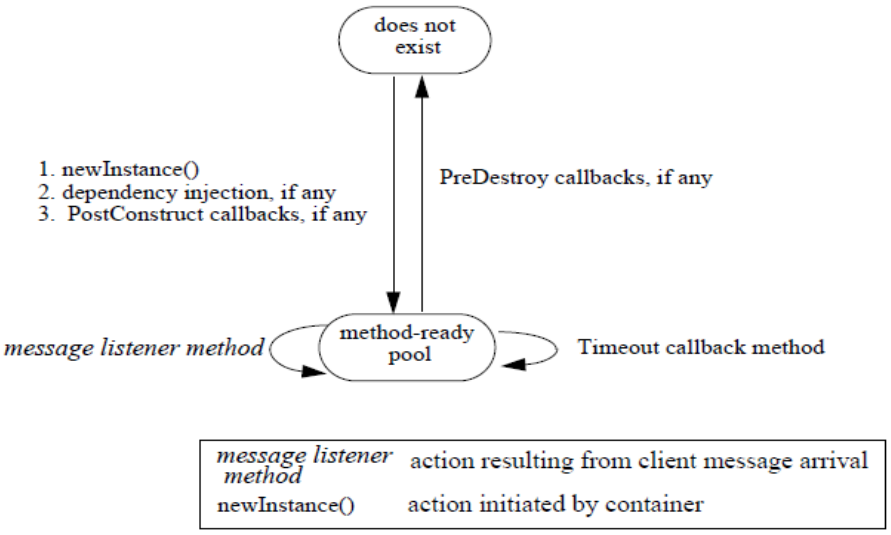
The source code for this application is in the *tut-install*/`javaeetutorial5/examples/ejb/simplemessage/` directory.

48. **MDB requirements**
    q. **MDB(S)** must be annotated with **@MessageDriven** , if there is no Deployment Descriptor
    r. The class must be PUBLIC
    s. The Class MUST not be ABSTRACT or FINAL
    t. MUST contain a PUBLIC constructor with NO arguments
    u. It MUST not DEFINE the finalize METHOD
    v. It is recommended , but not REQUIRED that a MDB class implements the MESSAGE LISTENER INTERFACE for the type it supports (For JMS ,javax.jms.MessageListener)
49. **MDB(S)** do not have LOCAL or REMOTE interfaces
50. **MDB** can be injected with **MessageDrivenContext** , this is commonly used to call **setRollbackOnly** method to handle EXCEPTIONS for a bean that uses **CONTAINER MANAGED TRANSACTIONS.** In addition to that to acquire ,

UserTransaction , **TimerService** , **getCallerPrincipal** , **isCallerInRole** are also available

51. **onMessage** is **CALLED** by the **CONTAINER** , **NOT** another **CLIENT**
52. By Using **RESOURCE INJECTION** and **ANNOTATION ejb-jar.xml** is **SKIPPED** , but at some situations **Application Server specific files may be needed to configure a MDB such as sun-ejb-jar.xml**
53. **CLIENTS DO NOT** access **MDB(S)** through **INTERFACES**
54. **MDB** has **ONLY** a **BEAN CLASS**, **NO INTERFACE DEFINED**
55. **MDB DOES NOT** retain any **DATA** or **CONVERSATIONAL STATE** specific to **CLIENT**
56. **ALL** instances of **MDB** are **EQUIVALENT**
57. **CONTAINER** can **POOL MDBs** to allow for a **STREAM of MESSAGES** to be **PROCESSED CONCURRENTLY**
58. A single MDB can process MESSAGES from MULTIPLE clients
59. **MDB(s)** are **RELATIVELY SHORT LIVED**
60. **MDB**(S) can be **TRANSACTION** aware
61. **MDB**(S) are **STATELESS**
62. A message can be DELIVERED to a MDB within a SINGLE TRANSACTION CONTEXT, so all operations within the onMessage method are PART of the TRANSACTION
63. If Message Processing ROLLSBACK , MESSAGE WOULD BE REDELIVERED
64. SESSION BEANS allow you to SEND JMS messages and RECEIVE those SYSNCHRONOUSLY , but not ASYNCHRONOUSLY
65. In General JMS messages should not be SENT SYNCHRONOUSLY

66. For **MDBs , PostContruct , PreDestroy** life cycle methods are supported
67. **PostConstruct** method is invoked before the first message listener method invocation on the bean
68. **PreDestroy** method is invoked at the time bean is removed or destroyed
69. **MDB(S)** message ACKNOWLEDGEMENT is automatically handled by the CONTAINER
70. **PreDestroy** call backs might be missed by the CONTAINER due to CONTAINER CRASHES etc.



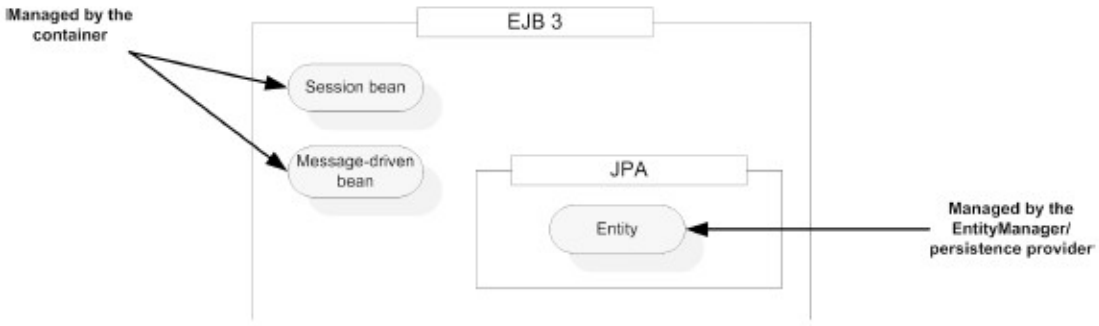
71.

ENTITY CLASSES – EJB 3.0 / ENTITIY BEANS – EJB 2.1

1. **EJB 3.0 ENTITY CLASSES** replace ENTITY BEANS in EJB 2.1
2. **ENTITY CLASS** is a **light** weight persistence DOMAIN Object
3. **ENTITY CLASSES** must be annotated with **@Entity** or use XML descriptors

4. **ENTITY CLASSES** must have a NO-ARGUMENT PUBLIC constructor
5. **ENTITY CLASS** must not be FINAL , no methods in class MUST be final
6. **ENTITY CLASS** must implement SERIALIZABLE interface if needs to be passed as a TRANSFER OBJECT to client side
7. **ENTITY CLASSES** can be ABSTRACT and EXTEND from NON-ENTITY
8. **ENTITY CLASSES** support POLYMORPHISM , INHERITANCE
9. **ENTITY** becomes PERSISTANCE by means of ENTITY MANAGER
10. **EVERY ENTITY MUST** have a primary KEY
11. relationships between ENTITIES could be
    w. **One-to-One**
    x. **One-to-Many**
    y. **Many-to-One**
    z. **Many-to-Many**
12. Relationship may be Bi-Directional , or Uni-Directional
13. Bi-Directional , has both **Owning** and **Inverse** side , Uni-Directional only Owning side
14. **ENTITY MANAGER** is **ASSOCIATED** with a **PERSISTANCE CONTEXT**
15. **PERSISTANCE CONTEXT** is a set of **ENTITY INSTANCES** in which for any persistence entity identity there is a **UNIQUE ENTITY INSTANCE**
16. **WITHING** the **PERSISTANCE CONTEXT** , entity instance **LIFE CYCLE** is managed
17. **ENTITY MANAGE** interface **DEFINES** the **METHODS** by which **PERSISTANCE CONTEXT** is **INTERACTED**

18. The **PERSIST** , **MERGE** , **REMOVE** , and **REFRESH** methods **MUST BE** invoked **WITHING** a **TRANSACTION CONTEXT** when used with an **ENTITY MANGER** which is **TRANSACTION SCOPED**
19. **FIND** , **GETEREFERENCE** methods **NO** need **TRANSACTION CONTEXT**
20.



21. **EJB 3.0 Entity Classes** CAN BE TESTED outside the CONTAINER

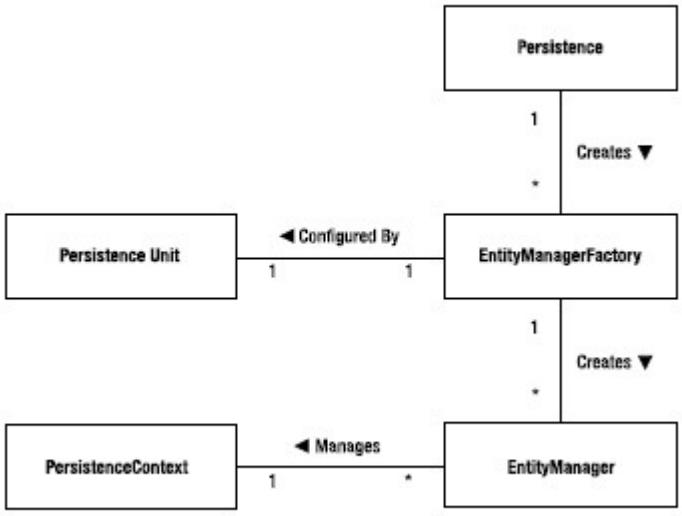


Figure 2-1. *Relationships between Java Persistence API concepts*

22.

23. **ENTITY MANAGER FACTORY** can be OBTAINED ,
    a. **EntityManagerFactory emf = Persistance.createEntityManagerFactory("Employee Service");**
24. **CREATING** an ENTITY MANAGER
    a. **EntityManager em = emf.createEntityManager();**
25. **PERSISTING ENTITIES**
    a. **Employee emp = new Employee(158);**
    b. **Em.persist(emp);**
26. **FIND and ENTITY**
    a. **Employee emp = em.find(Employee.class , 158);**
27. **REMOVING an ENTITY**
    a. **Employee emp = em.find(Employee.class,158);**
    b. **Em.remove(emp);**
28. **UPDATING an ENTITY**
    a. **Employee emp = em.find(Employee.class , 158);**
    b. **Emp.setSalary(emp.getSalary() + 1000);**

Entity Manager is not being invoked to do anything, but since Employee is a MANAGED entity, the changes must be saved automatically

29. **It is ASSUMED that** for PERSIST , REMOVE , MERGE , REFRESH are invoked in a TRANSACTION
30. **PERSISTANCE UNIT** is DEFINED in an XML configuration file named “persistence.xml”
31. Each PERSISTANCE UNIT has a UNIQUE NAME
32. A single PERSISTANCE UNIT XML file may contain MORE THAN ONE PERSISTANCE UNIT CONFIGURATIONS

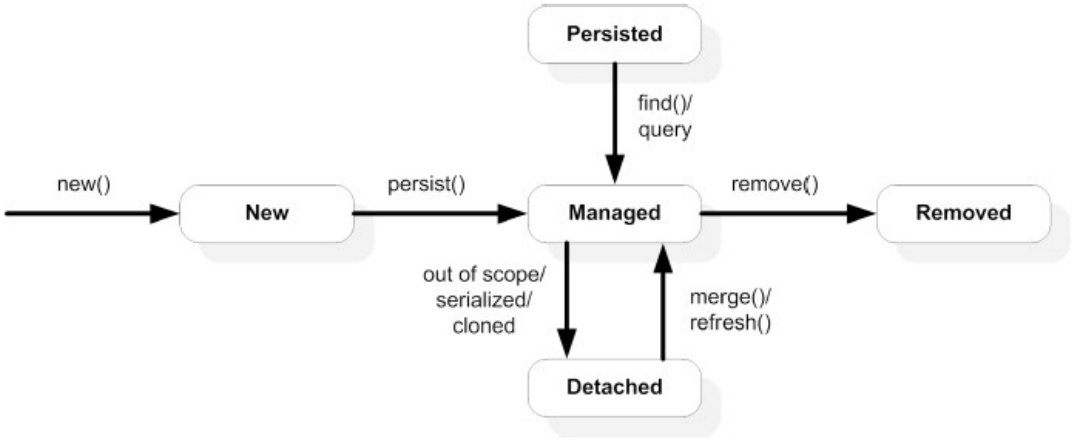
Listing 2-11. *Elements in the* `persistence.xml` *File*

```
<persistence>
    <persistence-unit name="EmployeeService" transaction-type="RESOURCE_LOCAL">
        <class>examples.model.Employee</class>
        <properties>
            <property name="toplink.jdbc.driver"
                      value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="toplink.jdbc.url"
                      value="jdbc:derby://localhost:1527/EmpServDB;create=true"/>
            <property name="toplink.jdbc.user" value="APP"/>
            <property name="toplink.jdbc.password" value="APP"/>
        </properties>
    </persistence-unit>
</persistence>
```

33.

34. persistence.xml file MUST be inside META-INF
35. The **transaction-type** attribute is used to specify whether the **ENTITY MANAGER** provided by entity manager factory for the **PERSISTANCE UNIT** must be **JTA** ENTITY MANAGERS **or RESOURCE LOCAL** entity managers
36. The **value** of **transaction-type** could be **JTA** or **RESOURCE_LOCAL**
37. A transaction type JTA assumes that a JTA data source will be provided
38. A transaction type RESOURCE_LOCAL assumes that a NON-JTA data source will be provided

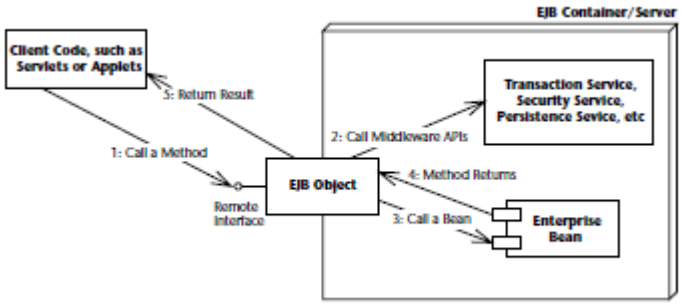
39. In a JAVA EE environment if the element is not specified the DEFAULT is JTA
40. In a Java SE environment if this element is not specified , a DEFAULT of RESOURCE_LOCAL will be assumed
41. Life cycle of a JPA class



42.

EJB 2.1 ENTITY BEANS

1. **EJB 2.1 ENTITY BEANS** are inherently COMPLEX
2. **LOT OF INTERFACECS** needed to be created / implemented
3. **LOT of LIFE CYLCE** related METHODS would appear in the CODE
4. Since EJB 2.1 ENTITY BEANS are distributed , remote objects PERISTANCE was introduced with a new problem and this cases PERFORMANCE issues
5. EJB 2.1 EJB OBJECT



6. **Figure 2.5** EJB objects.

7.

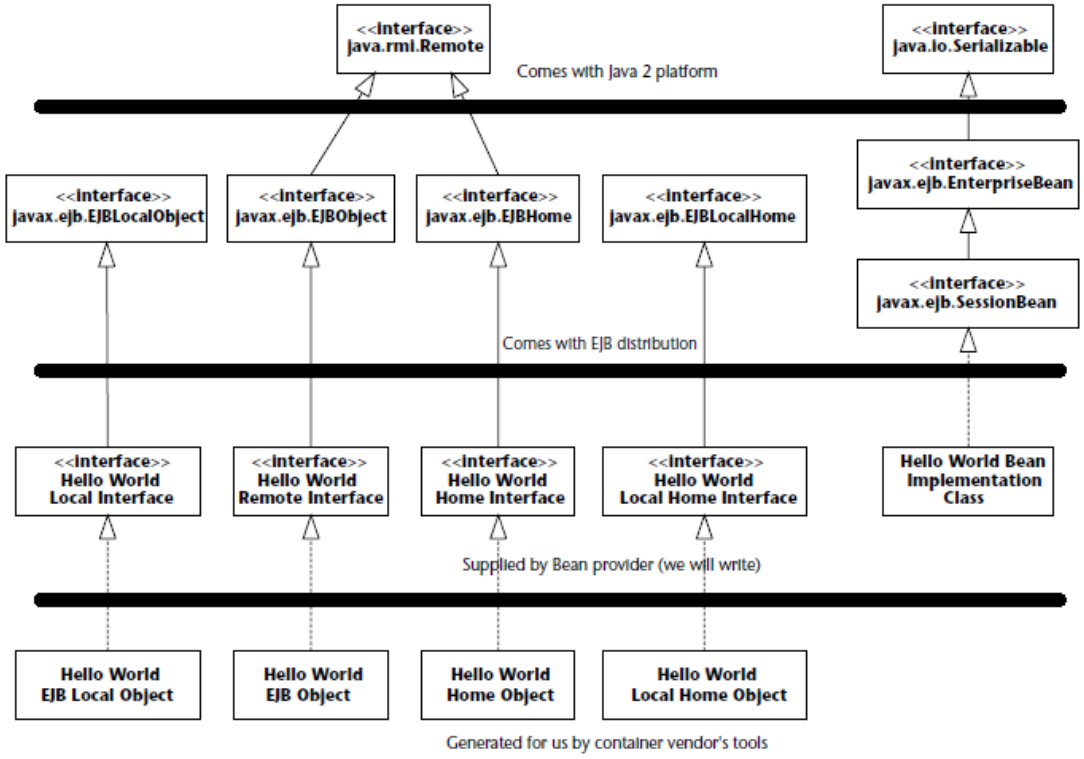


Figure 3.1 Our Hello world object model.

8. **Hello** remote Interface

a.

```
package com.ejb21;

import java.rmi.RemoteException;

public interface Hello extends EJBObject {
    public String hello() throws RemoteException;
}
```

9. **Hello** remote home Interface (This interface help in creating EJB remote interface)
    a.

```
package com.ejb21;

import java.rmi.RemoteException;

public interface HelloHome extends EJBHome {
    public Hello create() throws RemoteException,CreateException;
}
```

10. **Hello** enterprise java beans implementation

LIYANA ARACHCHIGE RANIL

a.

11. **ejb-jar.xml** file which binds all these together
    a.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterpris
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>Hello</ejb-name>
            <home>com.ejb21.HelloHome</home>
            <remote>com.ejb21.Hello</remote>
            <local-home>com.ejb21.HelloLocalHome</local-home>
            <local>com.ejb21.HelloLocal</local>
            <ejb-class>com.ejb21.HelloBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```

12. **Hello Client** using Hello Home Interface to create Hello Interface which gives the facility to interact with HelloBean implementation

a.

```
package com.ejb21;

import java.util.Hashtable;

public class HelloClient {

    public static void main(String[] args) throws Exception{

        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "org.jnp.interfaces.NamingContextFactory");
        env.put(Context.URL_PKG_PREFIXES,
        "org.jboss.naming:org.jnp.interfaces");
        env.put(Context.PROVIDER_URL,
        "jnp://127.0.0.1:1099");
        Context ctx = new InitialContext(env);
        Object obj = ctx.lookup("Hello");
        HelloHome home = (HelloHome) PortableRemoteObject.narrow(obj,HelloHome.class);
        Hello hello = home.create();
        System.out.println(hello.hello());
        hello.remove();
    }

}
```

13. **LocalHome** and **LocalEJBObject** is used to interact with EJB(S) within the same JVM

a.

```
package com.ejb21;

import javax.ejb.EJBLocalObject;

public interface HelloLocal extends EJBLocalObject {
    public String hello();
}
```

14. **Hello Local Home**

```
package com.ejb21;

import javax.ejb.CreateException;

public interface HelloLocalHome extends EJBLocalHome {
 public HelloLocal create() throws CreateException;
}
```

a.

15. **EJB 2.1 ENTITY** beans are PERSISTAT ENTITIES
16. **An ENTITY BEAN** does not PERFORM complex logic or workflows
17. **An ENTITY BEAN** consists of a set of standard classes , REMOTE INTERFACE , REMOTE HOME INTERFACE , ENTITY BEAN itself, deployment descriptor, LOCAL INTERFACE , LOCAL HOME INTERFACE
18. **ENTITY BEAN – REMOTE INTERFACE**
    a.

```
package com.ejb21.entity;

import java.rmi.RemoteException;

public interface Account extends EJBObject {
    public void deposit(double amt) throws RemoteException;
    public void withdraw(double amount) throws RemoteException;
    public double getBalance() throws RemoteException;
    public String getOwnerName() throws RemoteException;
    public String getAccountID() throws RemoteException;
    public void setAccountID(String id) throws RemoteException;

}
```

19. **ENTITY BEAN – REMOTE HOME**

a.

```
package com.ejb21.entity;

import java.rmi.RemoteException;

public interface AccountHome extends EJBHome {
 public Account create(String accountId,String ownerName) throws CreateException , RemoteException;
 public Account findByPrimaryKey(AccountPK key) throws FinderException,RemoteException;
 public Collection findByOwnerName(String name) throws FinderException,RemoteException;
 public double getTotalBankValue() throws RemoteException;
}
```

20. ENTITY BEAN – IMPLEMENTATION

LIYANA ARACHCHIGE RANIL

a.

```
package com.ejb21.entity;

import java.rmi.RemoteException;

public class AccountBean implements EntityBean {


    protected EntityContext ctx;

    public AccountPK ejbCreate(String accountID,String ownerName)
        PreparedStatement pstmt = null;
        Connection conn = null;

        try{
            System.out.println("EJB Create...");

            this.accountID = accountID;
            this.ownerName = ownerName;
            this.balance = 0;

            conn = getConnection();

            pstmt = conn.prepareStatement("insert into accounts (i
                    " values(?,?,?)");

            pstmt.setString(1,accountID);
            pstmt.setString(2,ownerName);
            pstmt.setDouble(3,balance);

            pstmt.executeUpdate();


        }catch(Exception e){
            e.printStackTrace();
        }finally{
            try{
```

21. ENTITY BEAN – DEPLOYMENT DESCRIPTOR

LIYANA ARACHCHIGE RANIL

a.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee" version="2.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xmls/ns/j2ee/ejb-jar_2_1.xsd">
    <display-name>AccountJAR</display-name>
    <enterprise-beans>
        <entity>
            <ejb-name>AccountEJB</ejb-name>
            <home>com.ejb21.entity.AccountHome</home>
            <remote>com.ejb21.entity.Account</remote>
            <ejb-class>com.ejb21.entity.AccountBean</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>com.ejb21.entity.AccountPK</prim-key-class>
            <reentrant>false</reentrant>
            <resource-ref>
             <res-ref-name>jdbc/bmp-account</res-ref-name>
             <res-type>javax.sql.DataSource</res-type>
             <res-auth>Container</res-auth>
             <res-sharing-scope>Shareable</res-sharing-scope>
            </resource-ref>
        </entity>
    </enterprise-beans>

    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>AccountEJB</ejb-name>
                <method-intf>Remote</method-intf>
                <method-name>*</method-name>
            </method>
            <method>
                <ejb-name>AccountEJB</ejb-name>
                <method-intf>Local</method-intf>
                <method-name>*</method-name>
```

22. ENTITY BEAN – LOCAL INTERFACE

```java
package com.ejb21.entity;

import javax.ejb.EJBLocalObject;

public interface AccountLocal extends EJBLocalObject {
    public void deposit(double amt);
    public void withdraw(double amt);
    public double getBalance();
    public String getOwnerName();
    public void setOwnerName(String name);
    public String getAccountID();
    public void setAccountID(String id);
}
```

a.

23. **ENTITY BEAN – LOCAL HOME INTERFACE**

a.

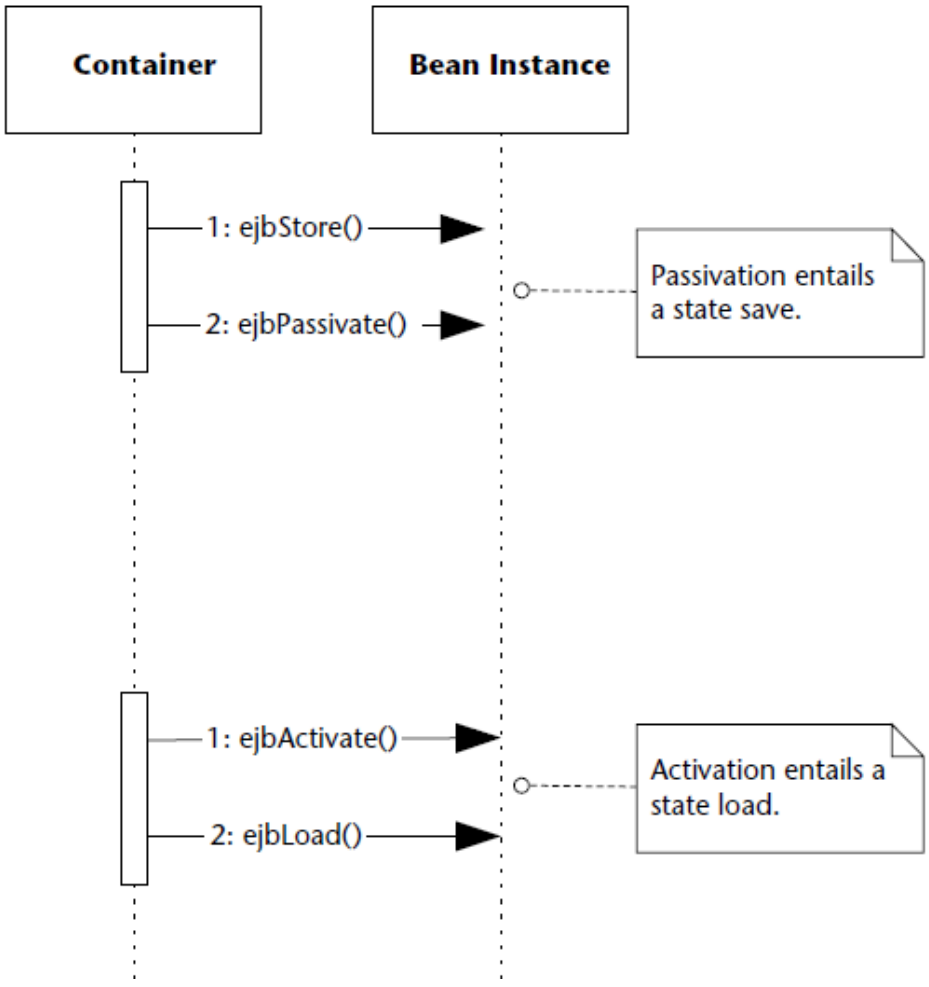
```java
package com.ejb21.entity;

import java.rmi.RemoteException;

public interface AccountLocalHome extends EJBLocalHome {
     public Account create(String accountId,String ownerName) throws CreateException;
     public Account findByPrimaryKey(AccountPK key) throws FinderException;
     public Collection findByOwnerName(String name) throws FinderException;
     public double getTotalBankValue();

}
```

24. **ENTITY BEAN** primary key MUST be SERIALIZABLE
25. **ejbLoad()** READS the DATA FROM persistence storage , **ejbStore()** PERSIST data to persistent storage
26. **ejbLoad() , ejbStore()** are invoked by the CONTAINER
27. CONTAINER may instantiates more than one instance of the SAME Entity Bean to serve more than one clients
28. To AVOID any data corruptions due to multiple ENTITY BEAN instances , container user ejbLoad() , ejbStore() call back

methods. At the end of TRANSACTIONS the Entity Beans will be synchronized with the data base

29. **CONTAINER MAY POOL** entity beans and ENTITY beans are RECYCLABLE (depends on the container POLICY)
30. **ENTITY BEANS** must implement **ejbActivate()** , **ejbPassivate()** container CALL BACKS
31. When an **ENTITY BEAN** is **PASSIVATED** , it **MUST** release it’s **RESOURCES** while it **MUST WRITE** the **STATE** to **DATABASE**

32.



33. **ENTITY BEAN** can be PERSISTED in TWO ways ,

a. **BEAN MANAGED PERSISTANCE** (The bean itself is responsible to manage the PERSISTANCE)
b. **CONTAINER MANAGED PERSISTANCE** (With container managed persistence no persistence logic is written by hand. Then container it self would generate those required code. Container does this by SUBCLASSING the entity bean)

34. **ENTITY CONTEXT** allows ENTITY BEAN To interact with the CONTAINER and it defined the ENVIRONMENT VARIABLES
35. finder METHODS are defined in REMOTE HOME and LOCA HOME BOTH (**ejbFind**()).
36. Only when BEAN MANAGED PERSISTANCE is used , you need to code FINDER METHODS explicitly. Or else COTNAINER managed PERSISTANCE is used those are created automatically
37. There must be at least one FINDER method in the HOME INTERFACE

a.

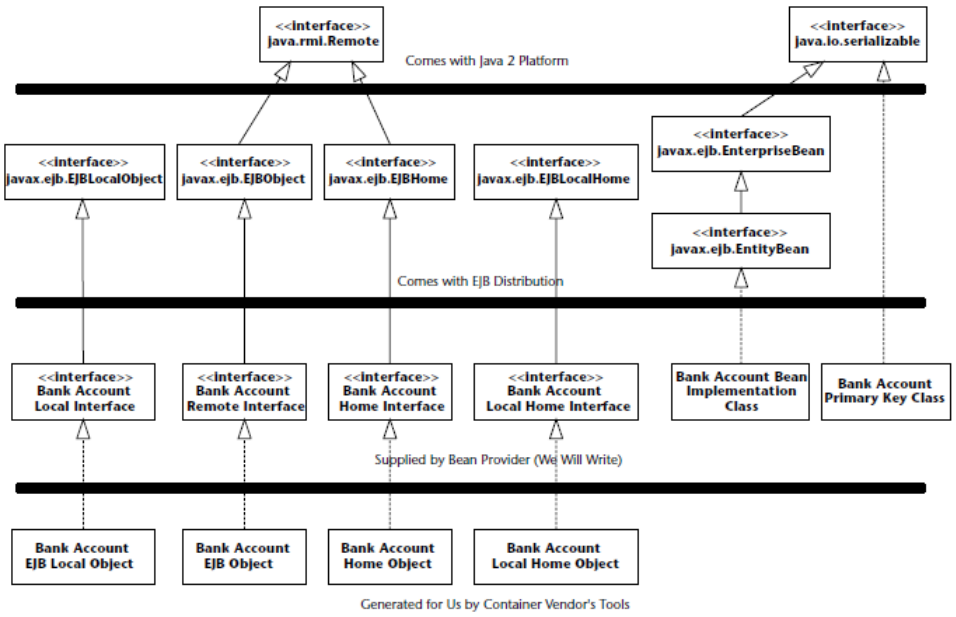
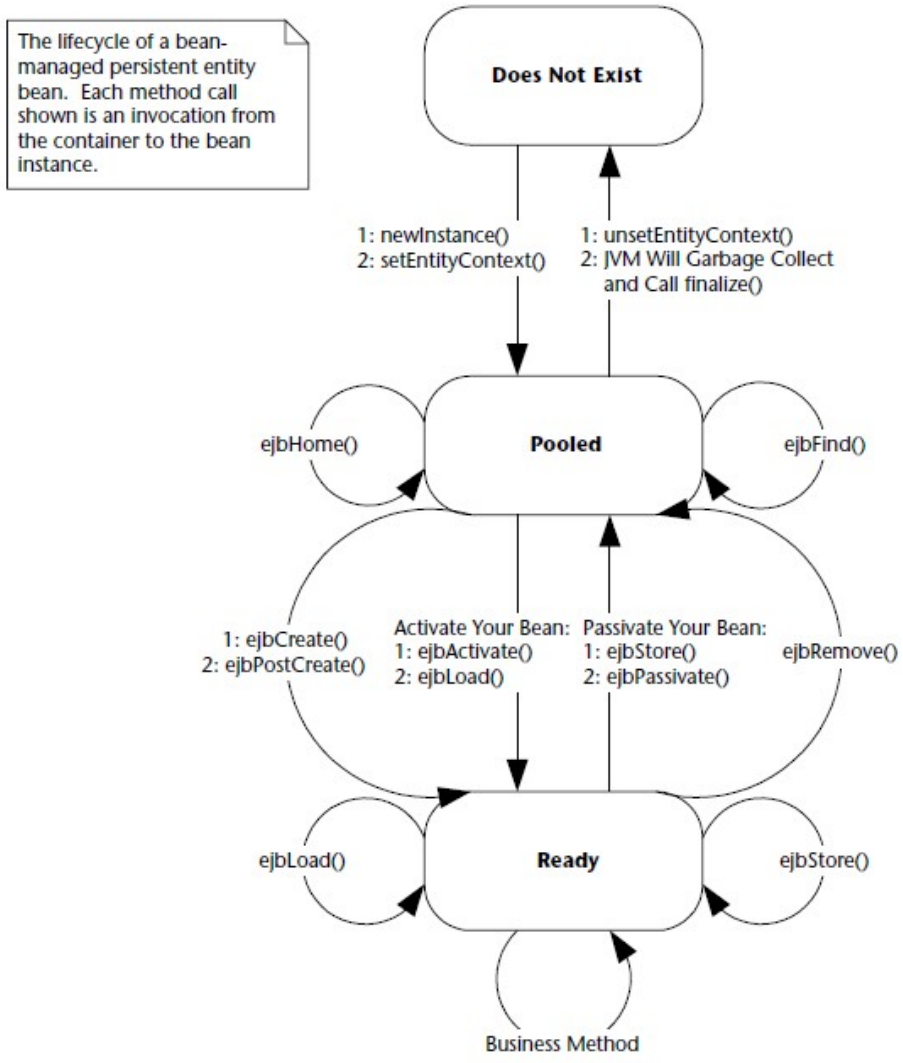


Figure 7.2 The bank account class diagram.

38. ENTITY BEAN life CYCLE

LIYANA ARACHCHIGE RANIL

a.



39. Container Managed Persistent (CMP) ENTITY BEAN SUPER CLASS

```
// CMP superclass
public abstract class CartBean implements EntityBean {
```

pter 8

```
// no fields

// abstract get/set methods
public abstract float getSubTotal();
public abstract float getTaxes();

// other business methods
public float getTotal() {
     return this.getSubtotal() + this.getTaxes();
}

// EJB required methods follow
```

a.

40. **CMP** ENTITY BEANS have an ABSTRACT PERSISTANCE SCHEMA

```
...
   <cmp-version>2.x</cmp-version>

   <abstract-schema-name>AccountBean</abstract-schema-name>

   <cmp-field>
    <field-name>accountID</field-name>
```

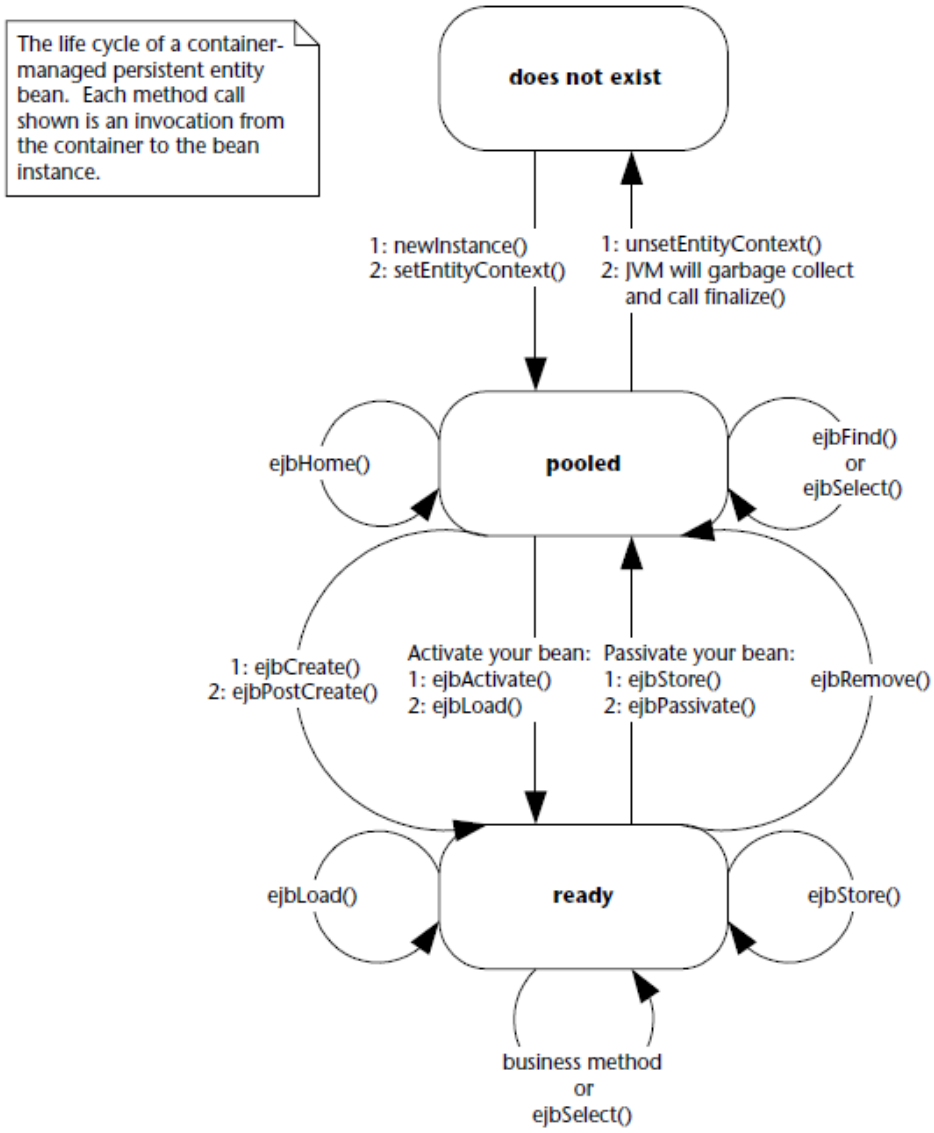
41.

42. **CMP ENTITY BEANS** have a QUERY language known as EJB-QL

43. ENTITY BEAN LIFE CYCLE – CMP

LIYANA ARACHCHIGE RANIL

a.



44. **CONTAINER MANAGED TRANSACTIONS = DECLARATIVE , BEAN MANAGED TRANSACTION = PROGRAMMATIC**

45. The benefit of PROGRAMMATIC TRANSACTION is you have all the control over the code that you write

46. The benefit of DECLARATIVE TRANSACTIONS is that you do not need to do much coding , CONTAINER will take care of it

47. TRANSACTIONS in EJB 2.1 ENTITY BEANS

a. When an ENTITY BEAN is invoked in a transaction it first call ejbLoad() to keep in sync with the DB
b. Then one or more BUSINESS METHODS are called
c. Then the TRANSACTION is COMMITED , then ejbStore() is invoked
d. The methods ejbLoad() , ejbStore() are invoked by the CONTAINER , not you
e. If we were to user Bean Managed Transaction we would need to write begin() , commit() methods inside the ENTITY BEAN. We could start the transaction in ejbLoad() and complete it in ejbStore(). But the problem is there is no guarantee about the time that these would be invoked
f. Hence Bean Managed Transaction is ILLEGAL for ENTITY BEANS (2.1)

48. **EJB 2.1 ENTITY BEANS , MUST** use CONTAINER MANAGED TRANSACTIONS (DECLARATIVE)

TRANSACTIONS

1. **TRANSACTIONS** have ACID properties , ATOMICITY , CONSISTANCY , ISOLATION , DURABILITY
2. **ATOMICITY –** either commits or rolls back together. Works as one unit
3. **CONSISTANCY –** If the system was in a consistent state before the transaction after the transaction is committed or rolled back it must also be in consistency
4. **ISOLATION –** Transactions do not step on one another
5. **DURABILITY –** Transactions once committed , must remain permanent
6. **TRANSACTION ISOLATION LEVELS**

a. **READ UNCOMMITED –** Transaction can read uncommitted data of other transactions
b. **READ COMMITED –** transaction can only read committed data by other transactions
c. **REPEATABLE READ –** transaction is guaranteed to get the same data for the same raw for multiple reads
d. **SERIALIZABLE –** Highest transaction isolation level , guarantees that the tables involved in this transaction will never be changed by other transactions

7. For DISTRIBUTED TRANSACTIONS , TWO PHASE COMMIT is used
8. **TWO PHASE COMMIT**
    a. – **PHASE 1** –Each participating resource manages coordinates local operations and forces all log records out. If successful , response with “OK”, if not allows timeout or send “OOPS”
    b. **– PHASE 2 –** If all participants responded “OK”, coordinator instruct all the participating resource manages to “COMMIT”. Participants COMMIT keeping LOG records. Or else coordinator instruct the participants to “ROLL BACK”

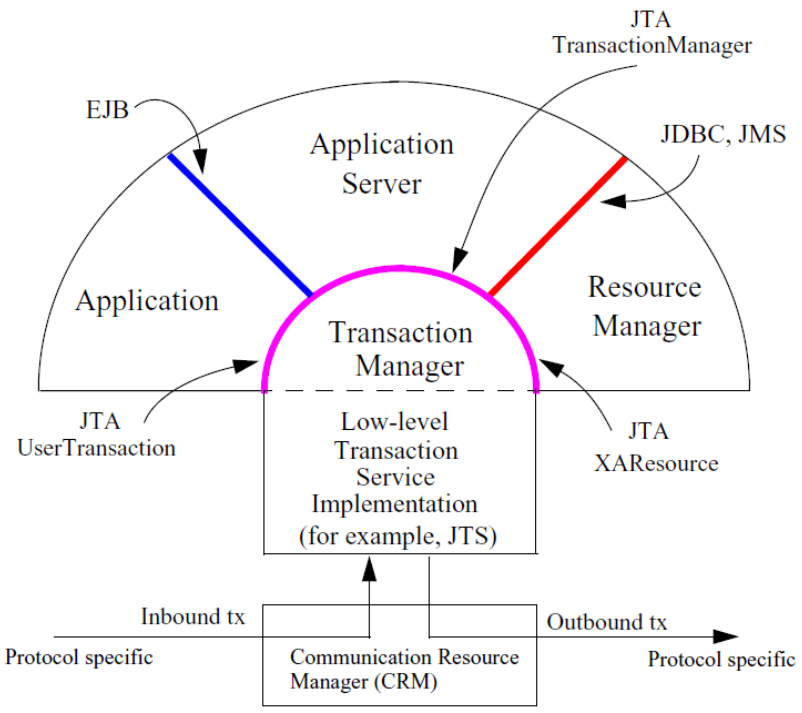
9.

Table 6.1 A transaction may be either local or global. A local transaction involves one resource and a global transaction involves multiple resources.

Property	Local	Global Transaction
Number of resources	One	Multiple
Coordinator	Resource Manager	Transaction manager
Commit protocol	Single-Phase	Two-phase

10. The **XA** protocol is the protocol that is used to talk to DIFFERENT TRANSACTION manages in case of TWO PHASE commit. This is developed by X/Open Group
11. A transaction may either be GLOBAL or LOCAL. A LOCAL transaction involves ONE resource while GLOCAL transactions involve MULTIPLE RESOURCES
12. Transaction Management support is provided in J2EE platform using JTA (Java Transaction API)
13. JTA defined application transaction server , interaction between application server , resource manager and transaction manager
14. JTA defines local JAVA interface BETWEEN a transaction manager and the parties involved in the transaction system : the application , the resource manager , application server
15. A TRANSACTION MANGER provides services and management functions required to support TRANSACTION DEMARCATION , TRANSACTION RESOURCE MANAGEMENT , SYNCHRONIZATION , TRANSACTION CONTEXT PROPAGATION

16. An APPLICATION SERVER provides infrastructure needed for the RUNTIME to support TRANSACTIONAL STATE MANAGEMENT. Such as EJB containers
17. A RESOURCE MANAGER provides the APPLICATION ACCESS to RESOURCE through the (RESOURCE ADAPTOR)
18. The RESOURCE MANGER participates in a DISTRIBUTED TRANSACTION by implementing a TRANSACTION RESOURCE INTERFACE used by the TRANSACTION MANGER. (Example resource manager is a Relational Database Systems)
19. Examples of RESOURCE ADAPTORS are JDBC drivers to connect to RELATIONAL DB , ODMG drivers to connect to Object Oriented Databases, JRFC to connect to SAP systems
20. A RESOURCE ADAPTOR is a library used by an APPLICATION SERVER or CLIENT to connect to a RESOURCE MANAGER



21.
22. Java Transaction Server (JTS) is the JAVA implementation of OMG Transaction Service on which JTA has been defined

23. **EJB** Requires that EJB COTNAINER supports application-level transaction demarcation by implementing the **javax.transaction.UserTransaction**
24. One of the NEW features included in JDBC 2.0 is support for DISTRIBUTED TRANSACTION. Two new interfaces have been created for JDBC drivers to support Distributed transactions using JTA's XAResource interface. The two new interfaces in JDBC 2.0 are **javax.sql.XAConnection** , **javax.sql.XADataSource**
25. **JTA** can be used by JMS service provides to support DISTRIBUTED TRANSACTIONS
26. **JMS** provider that support XAResource interface is able to participate as a RESOURCE MANAGER in a DISTRIBUTED TRANSACTION.
27. **JMS** provider would implement javax.transaction.xa.XAResource , javax.jms.XAConnection , javax.jms.XASession
28. **UserTransaction** interface provides the application the ability to control transaction boundaries programmatically
29. **EJB 2.1 TRANSACTIONS**
    a. **ENTITY BEANS** can only use DECLARATIVE (CONTAINER MANAGED) transactions
    b. **JMS** message driven beans can have **REQUIRED** and **NOTSUPPORTED** transaction attributes
    c. **JMS – if container managed transaction is used** JMS will read messages off from the destination in the same transaction as it performs the business logic. If something goes wrong transaction will be rolled back and message acknowledge will occur

d. **JMS – if BEAN managed transaction is used,** the transaction will start and end after JMS messages is received to the bean. Deployment descriptor acknowledgement modes should be used to acknowledge the message
e. **JMS – If no transaction is supported then ,** message acknowledgement happens at some point later after message is received by the bean
f. **JMS Point to Point Model** which uses QUEUE, the Reliability statement: A queue is typically created by an ADMINISTRATOR and last for long time. It is always available to HOLD messages SENT to it whether or not the CLIENT who CONSUMES messages is active or inactive. For this reason the client DOES NOT need to take any special precautions to insure that it DOSE NOT MISS messages
g. **JMS Publish / Subscriber** which users TOPIC, the Reliability statement: NON-DURALBLE subscriptions last for the lifetime of their subscriber object. This means a CLIENT will only see the messages PUBLISHED on a TOPIC while its SUBSRIBER is ACTIVE. If the SUBSCRIBER is not active it is MISSING messages published on the TOPIC
    i. **But at the cost of HIGHER overhead a SUBSCRIBER can be MADE DURABLE**
h. **EJB 2.1 TRANSACTION ATTRIBUTES –**
    i. **REQUIRED –** always run in a transaction , if one exists it uses that , or else create new one
    ii. **REQUIRESNEW –**starts a new one if there is no existing transaction. If there is one , then SUSPEND that transaction and create a new one and finish it before REVOKING the suspended one

iii. **SUPPORTES –** If there is a transaction existing it runs with the existing transaction , if there is no transaction then it just runs
iv. **MANDATORY –** Transaction must be existing , else exception is thrown
v. **NOTSUPPORTED –** If there is a transaction. It is SUSPENDED and the code will run without transaction. If there is no transaction then code just runs
vi. **NEVER –** If there is a transaction, then exception is thrown. Else the code just runs

i. For PROGRAMMATIC TRANSACTION in EJB 2.1 , JTA must be used

EJB 3.0 TRANSACTIONS

1. There can be CONTAINER MANAGED , BEAN MANAGED or CLIENT CONTROLLED TRANSACTIONS
2. CONTAINER MANAGED transactions allow components to be automatically enlist in TRANSACTIONS. EJB container takes care of everything
3. In EJB 3.0 you can specify the transaction attributes either via ANNOTATION or DEPLOYMENT DESCRIPTOR. @TransactionManagement annotation can be used
4. If neither the bean provider nor the deployer specifies transaction management , then the default is assumed to be container managed

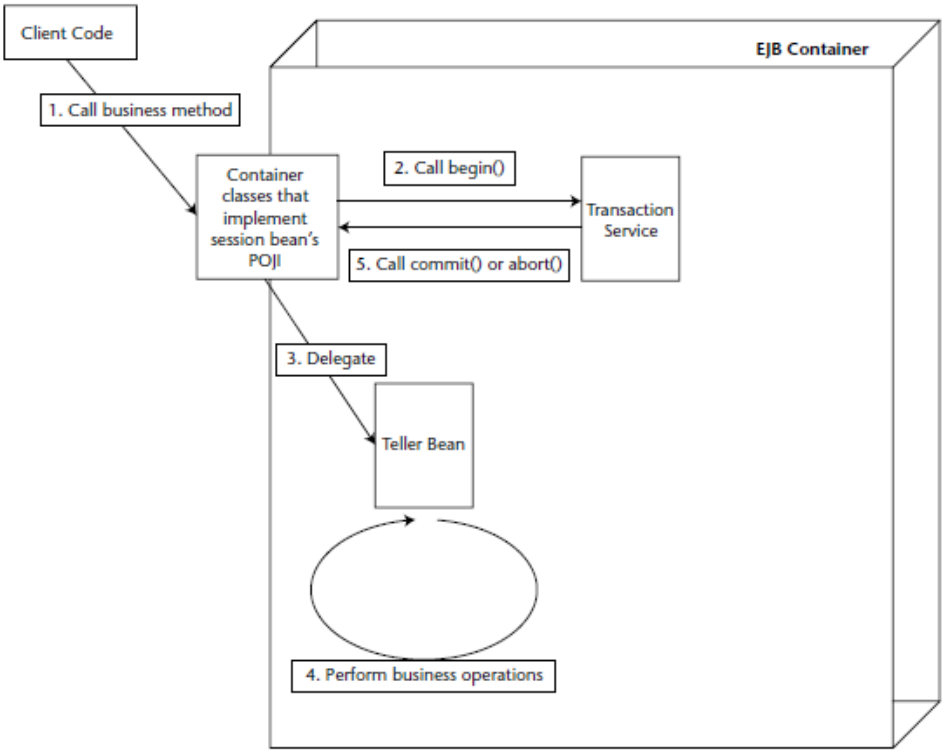


Figure 10.8 Container-managed transactions.

5.

6. Use code to start and end transactions outside of BEAN code. It is still needed to specify the transaction that is used by the EJB in this case

a.

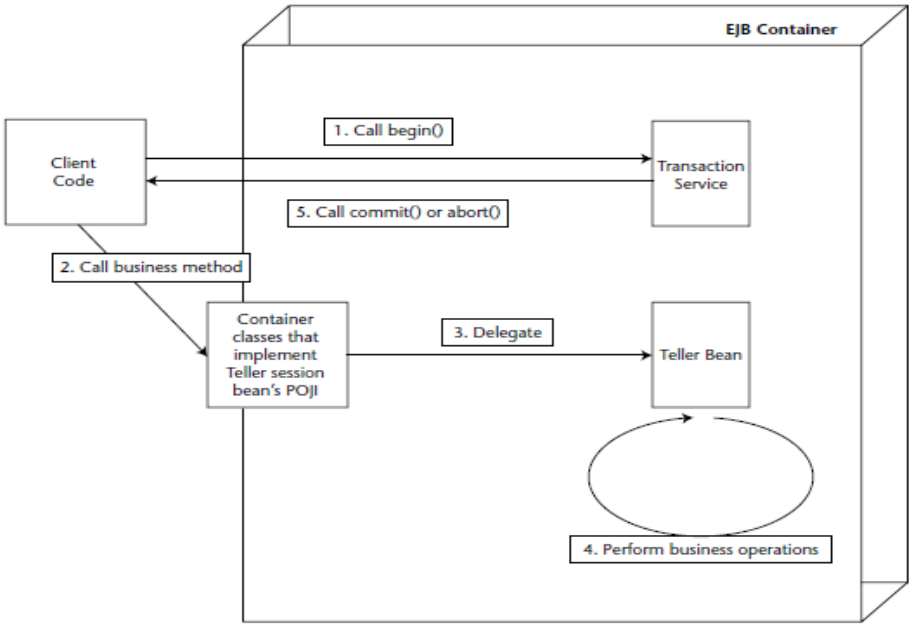


Figure 10.9 Client-controlled transactions.

7. BEAN MANAGED TRANSACTIONS

a.

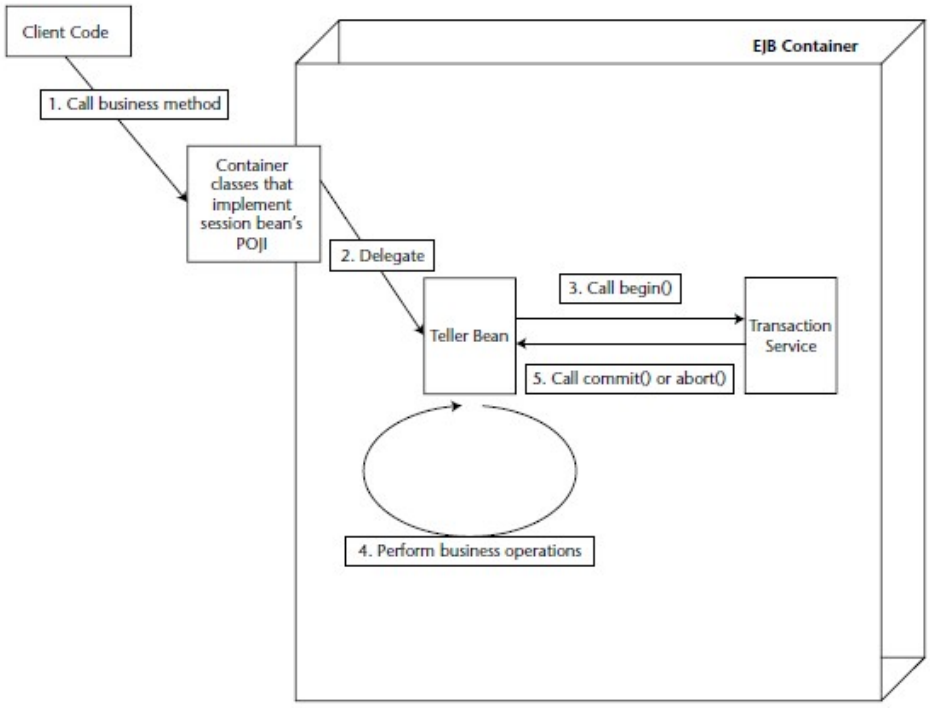


Figure 10.7 Bean-managed transactions.

8. CONTAINER MANAGED TRANSACTIONS – TRANSACTION ATTRIBUTES. IN EJB 3.0 @TransactionAttribute can be used
    a. **Required** – Bean will always have to run in a TRANSACTION. If there is no transaction then one is created. If one exist then that one is used
    b. **RequiresNew** – Always new transaction is created. If one exists it would be blocked and new one is started. If no existing transaction one will be started

c. **Supports** – If there is no transaction nothing is done , if there is one then still the method is executed without any issue
d. **Mandatory** – (Mandate a transaction which is running) If there is no transaction EXCEPTION is thrown. If there is a transaction then the execution happens in that transaction
e. **NotSupported** – Transactions are not supported, if one existing then it would be blocked until the execution of this method finishes
f. **Never** - If there is an existing transaction an EXCEPTION is thrown.
g.

```
import javax.ejb.*;
import javax.annotation.Resource;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;

@Stateless()
@TransactionManagement(javax.ejb.TransactionManagementType.CONTAINER)
public class TellerBean implements Teller {
    @PersistenceContext private EntityManager em;
    @Resource private SessionContext ctx;

    @TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
    public void transferFunds(float amount, String fromAccount, String
                              toAccount)
    {
        // Lookup for accts with the provided account IDs

        BankAccount acct1 = em.find(BankAccount.class, fromAccount);
```

9. **DOOMING** a transaction means force a transaction to ABORT. For this use **EJBContext's setRollbackOnly()** method
10. **STATELESS SESSION BEANS** support all the transaction attributes
11. A method on a **WEB SERVICE endpoint stateless** bean **CAN NOT** support **MANDATORY** transaction attribute
12. By Implementing SESSIONSYNCHRONISATION interface STATEFUL sessions beans can find information about the transaction that is it participating
13. **SessionSynchronization interface** can only be use with STATEFULL SESSION beans when those beans use CONTAINER MANAGED transactions
14. Client TRANSATION makes no sense for MDBs since those are not directly invoked by a CLIENT. Hence SUPPORTS, REQUIRESNEW, MANDATORY, NEVER will have no meaning. Only REQUIRED and NOTSUPPORTED are applicable
15. **BEAN MANAGED** transaction gives a more control than CONTAINER MANAGED

a.

```
import javax.ejb.*;
import javax.annotation.Resource;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.transaction.UserTransaction;

@Stateless()
@TransactionManagement(javax.ejb.TransactionManagementType.BEAN)
public class TellerBean implements Teller {
    @PersistenceContext private EntityManager em;
    @Resource private javax.transaction.UserTransaction userTx;

    public void transferFunds(float amount, String fromAccount, String
                              toAccount)
    {
        // Lookup for accts with the provided account Ids
        try {
            userTx.begin();

            BankAccount acct1 = em.find(BankAccount.class, fromAccount);
            BankAccount acct2 = em.find(BankAccount.class, toAccount);

            if (acct1.balance < amount)
                userTx.rollback();
```

16. For Programmatic TRANSACTION , you need to use UserTransaction interface in JTA
17. JPA supports both JTA and LOCAL TRANSACTIONS at the ENTITYMANAGERS level
18. There is no MECHANISM to specify the TRANSACTIONAL BEHAVIOR of the ENTITIES

19. When PERISISTANT API is used within a EE/EJB container , it is REQUIRED to support both JTA transactions as well as LOCAL transactions
20. The JTA transaction always begins and end external to the JTA entity Manager
21. The entity manager therefore only participate in an ACTIVE JTA transaction
22. For programmatic transactions use UserTransaction interface

```
public interface javax.transaction.UserTransaction
    public void begin();
    public void commit();
    public int getStatus();
    public void rollback();
    public void setRollbackOnly();
    public void setTransactionTimeout(int);
}
```

23.
24. When JPA is used in MANAGED environment such as inside a application server , for bean managed transactions use **UserTransaction** interface. This is assuming that that the managed environment supports JTA. If JPA is used stand alone without a managed environment , use **EntityTransaction** service to handle transactions
25. **TRANSACTIONS and JAVA EE CONNECTOR**
    a. Java connector architecture defines standard CONTRACT between RESOURCE ADAPTORS and Application servers
    b. This standard contract helps APPLICATION SERVERS to provider RUNTIME and INFRASTRUCTURE for transaction management of RA components
    c. RA (Resource Adaptor) can support Local Transactions as well as Distributed Transactions

d. If RA Supports LOCAL TRANSACTION then the client will have to acquire Common Client Interface (CCI) API object such as javax.resource.cci.LocalTransaction or an equivalent from the RA to demarcate the transaction
e. If RA supports Distributed Transaction the container will automatically enlist the client in the transaction context , if the client wants to work in a Distributed Transaction
f. Java EE connector architecture 1.5 supports the INFLOW of transactions from an EIS to the java EE environment
g. This will allow the Java EE application to participate in a Transaction initiated by EIS

26. **TRANSACTION ISOLATION**

a. ISOLATION is a guarantee that the concurrent users are isolated from one another
b. Choosing the correct level of ISOLATION is critical to robustness and scalability of the application
c. READ_UNCOMMITED : does not offer any isolation , uncommitted data is read. Provides highest performance
d. READ COMMITED : Only committed changes are read from the DB. This solves DIRTY READ problems
e. REPEATABLE READ : solves dirty reads as well as unrepeatable read issue
f. SERIALIZABLE : solves previous problems as well as PHANTOMS

27.

Table 10.5 The Isolation Levels

ISOLATION LEVEL	DIRTY READS?	UNREPEATABLE READS?	PHANTOM READS?
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

SECURITY

1. security comes at a price , such as INCREASED COST , COMPLEXITY ,REDUCED PERFORMACE , MAINTAINABILITY ,FUNCTIONALITY etc
2. WEB APPLICATION security is covered by Java Servlet Specification
3. Don’t try to roll your own security frameworks, algorithms etc. There are enough out there and must be able to use those
4. The general security concepts used by JAVA EE for both SERVLETS and EJB are very similar
5. AUTHENTICATION in WEB APPLICATIONS , there are few mechanisms supported
    a. HTTP BASIC – username and password is requested via system generated window. The values are passed to server as base 64 encoded values
    b. DIGEST AUTHENTICATION – username and password in transmitted in encrypted form. But this is not widely used

   c. FORM AUTHENTICATION – username and password is gathered via custom build forms and value are passed as plain text
   d. CLIENT CERT – using PKI x.509 certificates
6. Authorization for a JAVA EE WEB application could be done in two ways
   a. Declarative Security – Servlet container checks access to WEB RESOURCES based on the access rules in DEPLOYMENT DESCRITOR
   b. Programmatic Security –The Servlet performs its own checks based on internal state, hard coded access rules and authentication information provided by the container
7. CONFIDENTIALITY and INTERGRITY protection for WEB APPLICATIONS is based entirely on SECURE TRANSPORT which means HTTPS
8.

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>wholesale</web-resource-name>
      <url-pattern>/acme/wholesale/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
   </web-resource-collection>
   <auth-constraint>
      <role-name>CONTRACTOR</role-name>
   </auth-constraint>
   <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
   </user-data-constraint>
</security-constraint>
```

9. **user-data-constraint** contains **transport-guarantee** element that requires confidentiality protection from the TRANSPORT LAYER. Other values for the transport-guarantee are “**INTEGRAL**”[intergrity] and “**NONE**”. **CONFIDENTIAL** implies INTEGRAL automatically since CONFIDENTIAL ones are protected against modifications

10. The confidentiality and integrity protections for WEB APPLICATIONS are relatively COARSEGRAINED. There is no way for the DEPLOYMENT DESCRIPTOR to express requirement on the CRYPTOGRAPHIC STRENGTH of the protection through the CHOICE of SSL/TLS cipher suite
11. There are two security measures that CLIENTS must pass when you add security to an EJB system, AUTHENTICATION and AUTHORIZATION
12. AUTHENTICATION must be performed before any EJB method is called. AUTHORIZATION occurs at the beginning of each EJB method call
13. AUTHENTICATION logic can be called using JAVA AUTHENTICATION and AUTHORIZATION SERVICE (JAAS)
14. The system property java.security.auth.login.config is used to reference the resource containing the configuration information in JAAS.
    a. Java –Djava.security.auth.login.config=client.config
15. You will also NEED to SPECIFY SECURITY PERMISSION in order to execute the code (LoginContext lctx = new LoginContext(“helloclient”,new CallbackHandler)).
    a. Java –Djava.security.policy = client.policy
    b. The policies needed are
        i. Permission javax,security.auth.AuthPermission createLoginContext.SecurityExampleClient*;
        ii. Pemission javax.security.auth.AuthPermission “modifyPrivateCredentials”
16. AUTHORIZATION in EJB, after the client has been AUTHENTICATED, it must pass AUTHORIZATION test to call method on your bean. EJB Container ENFORCES authorization by

DEFINING SECURITY POLICIES for your BEAN. There are TWO ways

a. With PROGRAMMATIC AUTHORIZATION, you hard code security checks into your bean code
b. With DECLARATIVE AUTHORIZATION , the CONTAINER performs all AUTHORIZATION checks for you

17.
```
<role-name>administrators</role-name>
```

```
@Stateless
@DeclareRoles({"administrators"})
public class EmployeeManagementBean {
    ...
}
```

18. All security checks are made possible due to SECURITY CONTEXT.
19. SECURITY CONTEXT encapsulate the CURRENT CALLERS security STATE
20. CONTAINER uses SECURITY CONTEXT behind the scene
21. You can control the way that SECURITY INFORMATION is PROPAGATED via ANNOTATIONS or in your DEPLOYMENT DESCRIPTOR
22. If there is no EXPLICIT specification , either by DESCRIPTOR or ANNOTATION the caller PRINCIPLE is PROPAGATED

23.
```
<security-identity>
     <run-as>
          <role-name>admins</role-name>
     </run-as>
</security-identity>
```

```
@RunAs("admins")
@Stateless
public class EmployeeManagementBean {
    ...
}
```

EJB TIMER SERVICE

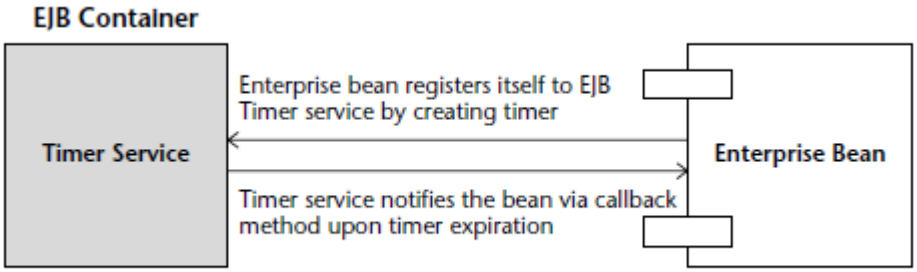
1. **EJB 2.1** introduced support for SCHEDULING through the CONTAINER-MANAGED EJB TIMER SERVICE
2. **Enterprise BEANS** interested in receiving TIMER NOTIFICATIONS will REGISTER themselves with the TIMER SERVICE
3. **STATELESS SESSION BEANS ,** ENTITY BEANS , MESSAGE DRIVEN BEANS can receive TIMED notifications from the CONTAINRE
4. **STATEFUL SESSION BEANS , JAVA PERSISTANT ENTITIES** DO NOT SUPPORT TIMERS
5. 

Figure 12.1 Interaction between the Timer Service and EJB.
6. **TimerService** instane can be accessed through EJBContext

7.
```
@Timeout
public void handleTimeout(Timer timer)
{
  System.out.println("CleanDayLimitOrdersBean: handleTimeout
  called.");

  // Put here the code for cleaning the database of day limit orders
  // that have not been executed.
}
```
8. Since EJB class does not allow static variables a TRUE SINGLETON can not be written. But as workarounds LIMIT THE POOL SIZE , USE RMI-IIOP and JNDI
9. WHEN TO USE MESSAGING versus RMI-IIOP
    a. **Database Performance** – can process messages at OFF PEAK db load hours
    b. **Quick Response** – Client may not want to block , ASY messaging
    c. **Smoother Load Balancing** – With session and entity beans , load balancing algorithms make educated guesses about which server is the least loaded,with messaging the server that is the least loaded will ASK for a message
    d. **Request Prioritization –** Asynchronous servers can QUEUE , PRIORITIZE and process messages in a different order that that in which they arrived into the system
    e. **Rapid integration of disparate systems –** Many legacy systems are based on Message Oriented middleware and can easily interact with JAVA EE system through MESSAGING
    f. **Loosely Coupled Systems –** Messaging enable loosely coupled systems
    g. **Geographically Disperse Systems –** Messaging is very useful when you have applications communicating over the Internet or a wide area network

h. **Reliability –** Messaging can be used even if the server is down
i. **Many to Many communication –** Messaging is appropriate since it enables many producers and many consumers
j. **When You are not sure if the operation succeeds –** RMI-IIOP can throw Exceptions , but MDB(S) can not
k. **When a return result is needed –** RMI – IIOP systems can return a results immediately
l. **When you need an operation to be part of larger transaction –** RMI-IIOP
m. **When you need to propagate client's security identity to the server –** RMI-IIOP
n. **When you are concerned about REQUEST PERFORMANCE –** Messaging is inherently slower since there is a middle man

EJB and INTEGRATION

1. There are THREE ways to integrate EJB applications
    a. JMS and JMS based Message Driven Beans
    b. Java Web Services
    c. Java EE Connector Architecture
2. **RESOUCE ADAPTORS (RA)** when deployed in a MANAGED environment such as an APPLICATION SERVER , the RA accepts REQUESTS from various JAVA EE Components, such as SERVLETS , JSPs , EJBs and translate those REQUESTS to EIS-SPECIFIC calls and sends those REQUESTS to EIS. The response that is received from EIS is forwarded to the client JAVA EE component

3. Application Components thus interact with the RA through its CLIENT CONTRACT
4. RA can support CLIENT CONTRACT using Common Client Interfaces (CCI) or EIS Specific client interfaces
5. RA can also work in a NON MANAGED environment, Ex- using JDBC driver in a non managed environment

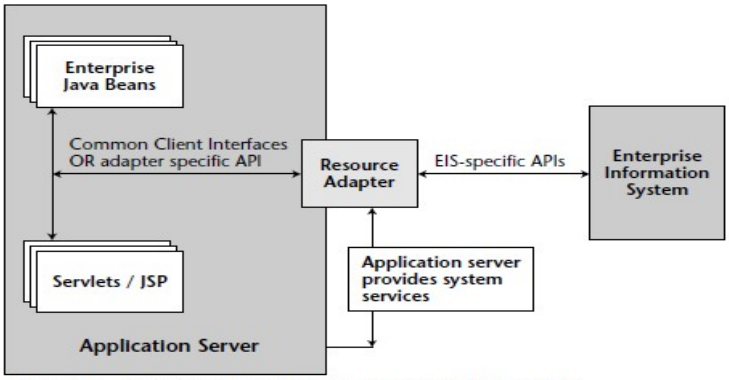


Figure 15.3 Interaction between the Java EE connector, RA, and EIS.

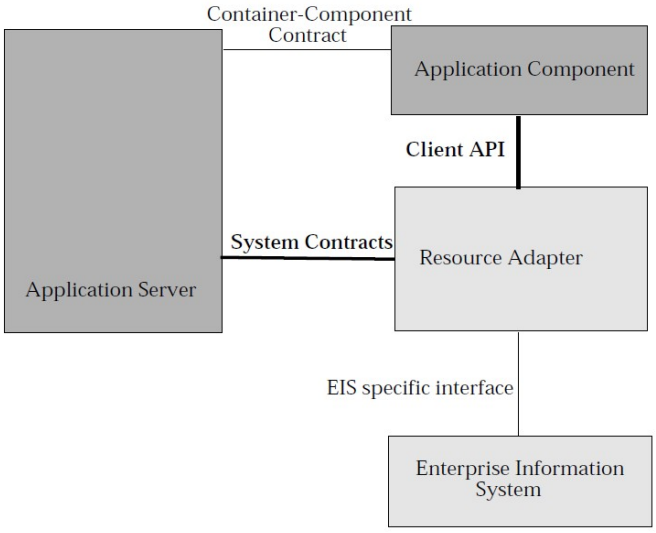
6.

7. RA has contracts with Application Server
    a. **Connection Management Contract –** enables application components to connect to the EIS so that application server can pool these connections
    b. **Transaction Management Contract –** allows transactional access to the EIS from your application component
    c. **Security Contract –** enable secure access to the EIS from the application component
    d. **Lifecycle management contract(1.5) –** allows the application server to manage the lifecycle functions
    e. **Work management contract(1.5) –** RA can submit work it needs to perform to the application server

f. **Transaction inflow contract (1.5) –** Allows RA to PROPOGATE the transaction context imported from the EIS to the Application Server. This contract supplements TRANSACTION MANAGEMENT CONTRACT

g. **Message inflow contract (1.5) –** Allows RA to ASYNCHRONOUSLY deliver messages to message ENDPOINTS residing in the APPLICATION SERVER

8.

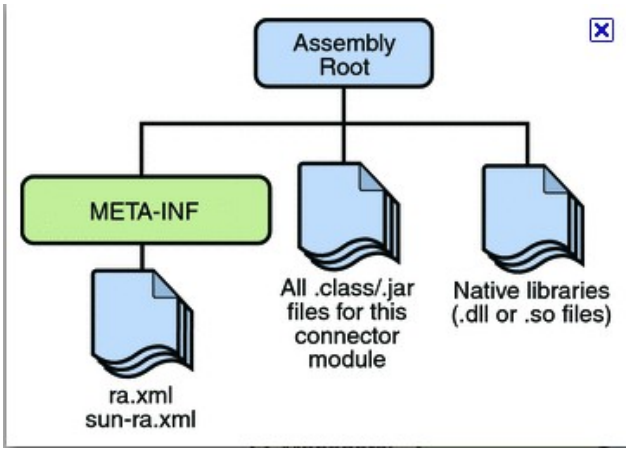


9.

10. **JCA** has different contracts between different participants
11. **Application Server** and Application Component , it has **Container-Component Contract (**this is Not by JCA **)**
12. **Application Server** and Resource Adaptor itself has a **System Contract**
13. **System Contract** has standard SET of system level contracts (Explained above)
14. The RA and Application Server collaborate to provide the underlying mechanisms , such as TRANSACTIONS , SECURITY

,CONNECTION POOLING ,and to DISPATCH to application components

15. The Client API used by APPLICATION COMPONENTS for accessing EIS could be CCI (Common Client Interface) or A Client API specific to the type of the RESOURCE ADAPTOR and its underlying EIS
16. The CONNECTOR ARCHITECURE requires that the connector architecture compliant RESOURCE ADAPTOR and APPLICATION SERVER must support SYSTEM CONTRACT
17. The Connector Architecture RECOMMENDS , though it is NOT MANDATED , that a RESOURCE ADAPTOR supports Common Client Interface (CCI) as CLIENT API
18. MANAGED clients as well as NON-MANAGED CLIENTS (Applets , Java applications) can use JCA to connect to EIS
19. Application container and RA has a SECURITY contract which does not depend on specific security technology implementation. The security can be enabled by enforcing Authentication and Authorization when connections are made to the EIS via RA
20. There are two ways , **Application** (From the Component CODE)can specify security details or it could be delegated to the **Application Container** (Ex: **Deployer sets up Username and Password to login to EIS**)to do the security authentication again EIS
21. The above two methods are also know as CONTAINER MANAGED SIGN ON and APPLICATION MANAGED SIGN ON
22. **AUTHENTICATION MECHANISM :** Application server and EIS collaboratively ensure that resource PRINCIPALS are properly AUTHENTICATED when the PRINCIPLE connects to EIS
23. Commonly supported AUTHENTICATION mechanisms in CONNECTOR ARCHITECTURE are BasicPassword,Kerbv5

24. **AUTHORIZATION MECHANISM :** Authorization checking to ensure that the principle has access to EIS resource can be applied at EIS or at the APPLICATION SERVER
25. **SECURE ASSOCIATION :** The communication between the Application Server and EIS is protected. The RA can use any SECURITY MECHANISM to establish the SECURE ASSOCIATION (GSS API is an EXAMPLE)
26. The connector architecture requires that the application server and the resource adaptor MUST support JAAS Subject class as part of its security contract.
27. However it recommends but does not mandate JAAS pluggable authentication modules
28. The connector architecture does not require support for the AUTHORIZATION portion of the JAAS framework
29. The resource adaptor can use MESSAGE INFLOW contract to call a MDB
30. 

WHEN TO USE JMS / WEB SERVICE / RA

1. **JMS**

a. Application wants to Integrate with your EJB application in ASYNCRONOUS yet reliable manner
b. Integrating Non REAL time applications –processing INVENTORY , SHIPPPING or COMMUNICATION with suppliers
c. Need RELIABILITY and TRANSACTION support

2. **JAVA EE CONNECTORS**
    a. Want to integrate with back end EIS application without MODIFYING them
    b. Quality of service is a PREREQUSITE for Integration – Transactional and SECURE , POOL outbound connections , application server needs to consume messages from EIS
    c. Integrating with a widely used EIS
3. **WEB SERVICE**
    a. Need to QUICKLY integrate application end points
    b. Target application for integration exist on DISPARATE PLATFORM
    c. Target application endpoints are deployed BEHIND the BMZ , needing to go through FIREWALL

EJB 3.0 vs EJB 2.1

1. The number of classes needed in EJB 3.0 is very much less than the number of classes needed in EJB 2.1. Ex, if you need to create Address , Customer and Subscription beans ,
    a. EJB2.1
        i. AddressBean
        ii. LocalAddress
        iii. LocalAddressHome

iv. CustomerBean
v. LocalCustomer
vi. LocalCustomerHome
vii. SubscriptionBean
viii. LocalSubscription
ix. LocalSubscriptionHome

b. EJB 3.0
i. Address
ii. Customer
iii. Subscripton

2. The class Definition are simpler in EJB 3.0 than it is in EJB 2.1. An Entity in EJB 3.0 is a Plain Old Java Object (POJO), no BOLIERPLATE code is required

a. **Address.java (EJB 3.0)**

```
@Entity
public class Address implements java.io.Serializable{
        public Address(){}
}
```

b. **AddressBean.java** / Container Managed Persistence **(EJB 2.1)**

```
public abstract class AddressBean implements EntityBean{
        public void setEntityContext(EntityContext ctx){
        }
        public void unsetEntityContext(){
        }

        public void ejbRemove(){}
        public void ejbLoad(){}
        public void ejbStore(){}
        public void  ejbPassivate(){}
        public void ejbActivate(){}
}
```

3. EJB 3.0 relies on ANNOTATIONS and that minimizes what needs to be specified
4. EJB 3.0 default values make things easier
5. Persistent FIELD declaration in EJB 3.0 is easier than EJB 2.1
    a. AddressBean.java **(EJB 2.1)**

```
public abstract String getAddressID();
public abstract void serAddressID(String id);

//XML deployment descriptor
<ejb-jar>
        <display-name>Ejb1</display-name>
        <enterprise-bean>
                <entity>
                        <ejb-name>AddressBean</ejb-name>
                        <cmp-field>
                                <field-name>addressID</fied-
        name>
                        </cmp-field>
                </entity>
        </enterprise-bean>
</ejb-jar>
```

    b. Address.java **(EJB 3.0)**

```
private String addressed;

public Address(String id){
        setAddressID(id);
        setStreet(street);
}

@column(name=”addressID”)
public String getAddressID(){
```

```
}

//No XML Descriptor Needed
```

6. Specifying ENTITY IDENTITY in EJB 3.0 is SIMPLER than in EJB 2.1
   a. XML Descriptor Needed **(EJB 2.1)**

```
<ejb-jar>
    <display-name>Ejb1</display-name>
    <enterprise-bean>
        <entity>
            <ejb-name>AddressBean</ejb-name>
            <cmp-field>
                <field-name>addressID</fied-
        name>
            </cmp-field>
            <prim-key-
            class>java.lang.String</prim-key-
            class>
            <primarykey-
            field>addressID</primarykey-field>

        </entity>
    </enterprise-bean>
</ejb-jar>
```

   b. Address.java (Can specify composite keys using @IdClass , @EmbeddedId) – **EJB 3.0**

```
@id
public String getAddressID(){
    return addressed;
}
public void setAddressID(String id){
    This.addressID = id;
}
```

7. Relationship mapping in EJB 3.0 is very much easier than that in EJB 2.1

EJB 2.1

```
public abstract class CustomerBean implements EntityBean{
    public abstract Collection getAddresses();
    public abstract void setAddresses(Collection addresses);

    public abstract Collection getSubscriptions();
    public abstract void setSubscriptions(Collection subscription);
}
```

//XML Descriptor

```
<relationships>
    <ejb-relation>
        <ejb-relationship-role>
            <ejb-relatinship-role-name>
                CustomerBean-AddressBean
            </ejb-relatinship-role-name>
            <multiplicity>One</multiplicity>
            <relationhip-role-souce>
                <cmr-field>
                    <cmr-field-name>addresses</cmr-field-name>
                    <cmr-field-type>java.util.Collection</cmr-f
                </cmr-field>
            </relationhip-role-souce>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relatinship-role-name>
                AddressBean-CustomerBean
            </ejb-relatinship-role-name>
            <multiplicity>Many</multiplicity>
            <cascade-delete/>
            <relationshio-role-source>
                <ejb-name>AddressBean</ejb-name>
            </relationshio-role-source>
        </ejb-relationship-role>
        ......
        ...
        ..
    </ejb-relation>
</relationships>
```

EJB 3.0

```
@Entity
Public class Customer implements java.io.Serializable{

@OneToMany(cascade=CascadeType.ALL,fetch=FetchType.EAGER)
        public Collection <Address> getAddresses(){
                return addresses;
        }

        @ManyToMany(fetch=FetchType.EAGER)
        @JoingTable(name=”CUSTOMERBEANSUBSCRIPTIONBEAN”
        joinColumns=@JoinColumn(name=”CUSTOMERBEAN_CUSTOME
        RID96”,referencedColumnName=”customerId”),
        inverserJoinColumns@JoinColumn(name=”SUBSCRIPTIONBEAN_
        TITLE”,referencedColumnName=”TITLE”))
        public Collection<Subscription> getSubscriptions(){
                return subsciptions;
        }
}

---ENTITY CLASS
@Entity
Public class Subscription implements java.io.Serializable{
        @ManyToMany(mappedBy=”subscription”)
        Public Collection<Customer> getCustomer(){
                return custimers;
        }
}
```

8. EJB 3.0 supports INHERITANCE and POLYMORPHISM which EJB 2.1 does not support that. You can map hierarchy of entities , where entity subclasses another , to a relational database structure, and submit QUERIES against the base class. Queries are treated polymorphically against entire class hierarchy

9. In EJB 3.0 Entities can INHERIT from other ENTITIES and from NON-ENTITIES
10. Operations on ENTITIES in EJB 3.0 is simpler than that of EJB 2.1. Entity operations are performed directly on the entity itself in EJB 3.0. in EJB2.1 JNDI was involved heavily
11. In EJB 3.0 , DEPENDENCIES can be INJECTED unlike in EJB 2.1. In EJB 2.1 for that JNDI was used
12. In EJB 3.0 TRANSACTION related specification are simplified than in EJB 2.1. EJB 2.1 needed XML descriptors while EJB 3.0 uses simplified ANNOTATIONS
13. Support for QUERIES has been significantly ENHANCED in EJB 3.0 , than EJB 2.1
14. Testing entities OUTSIDE the CONTAINER in EJB 3.0 is easier, EJB 2.1 did not allow this possibility due to HOME and remote interfaces