CMR
INSTITUTE OF
TECHNOLOGY

CMRIT
CELEBRATING 25 YEARS
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 3 – Answer key

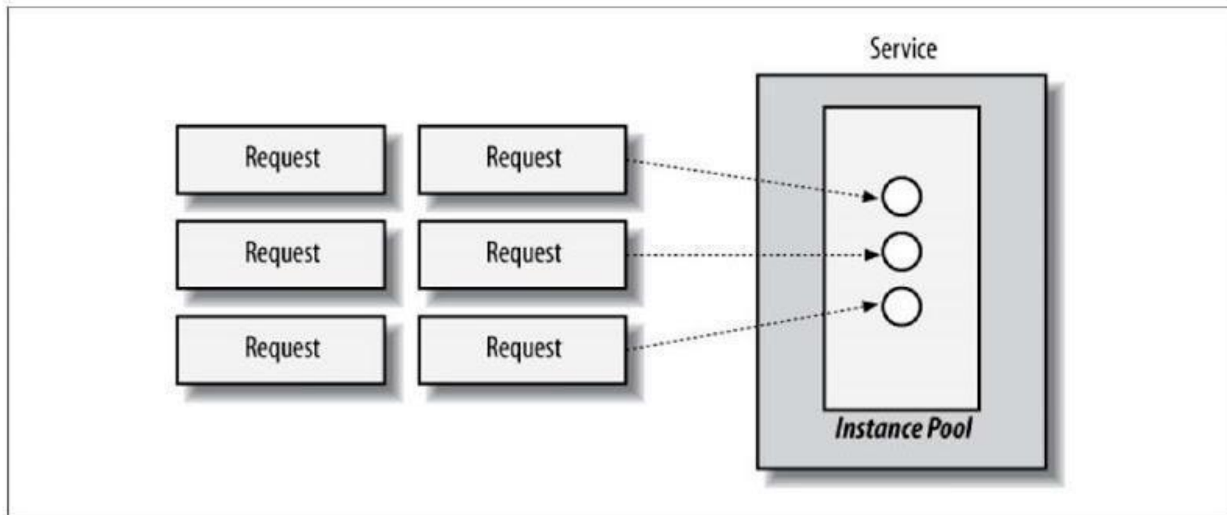
Sub:	Advanced Java Programming							Sub Code:	18MC A41
Date:	19/06/2021	Duration:	90 min's	Max Marks:	50	Sem	4	Branch:	MCA

1a(Explain the container services provided by component mode of EJB

Dependency Injection
2. Concurrency
3. Instance Polling and Caching
4. Transactions
5. Security
6. Timers
7. Naming and Object Stores, JNDI
8. Lifecycle callbacks
9. Interoperability
10. Interceptors

1. Instance Pooling/Caching

Because of the strict concurrency rules enforced by the Container, an intentional bottleneck is often introduced where a service instance may not be available for processing until some other request has completed.
If the service was restricted to a singular instance, all subsequent requests would have to queue up until their turn was reached



EJB addresses this problem through a technique called instance pooling, in which each odule is allocated some number of instances with which to serve incoming requests Many vendors provide configuration options to allocate pool sizes appropriate to the work being performed, providing the compromise needed to achieve optimal throughput.

2. Transactions

Transactions provide a means for the developer to easily delegate the creation and control of transactions to the container.

 When a bean calls createTimer(), the operation is performed in the scope of the current transaction. If the transaction rolls back, the timer is undone and it's not created

 The timeout callback method on beans should have a transaction attribute of RequiresNew.

 This ensures that the work performed by the callback method is in the scope of container-initiated transactions.

3. Security

Most enterprise applications are designed to serve a large number of clients, and users are not necessarily equal in terms of their access rights.

An administrator might require hooks into the configuration of the system, whereas unknown guests may be allowed a read-only view of data.

If we group users into categories with defined roles, we can then allow or restrict access to the role itself, as illustrated in Figure 15-1.

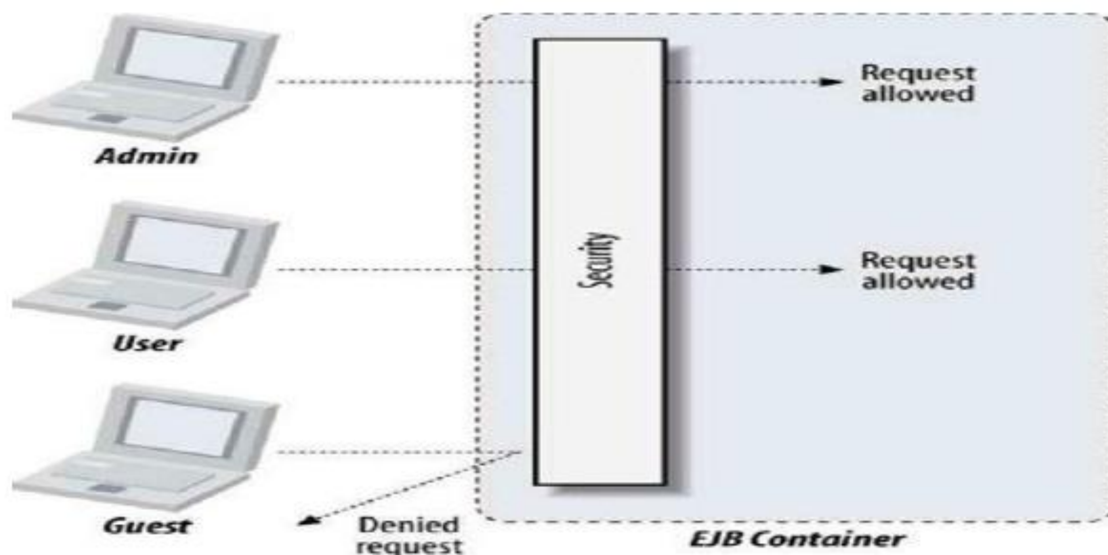


Figure 15-1. EJB security permitting access based upon the caller's role

This allows the application developer to explicitly allow or deny access at a fine-grained level based upon the caller's identity

4.Timers

We dealt exclusively with client-initiated requests. While this may handle the bulk of an application’s requirements, it doesn’t account for scheduled jobs:

• A ticket purchasing system must release unclaimed tickets after some timeout of inactivity.

• An auction house must end auctions on time.
• A cellular provider should close and mail statements each month.

The EJB Timer Service may be leveraged to trigger these events and has been enhanced in the 3.1 specification with a natural-language expression syntax.

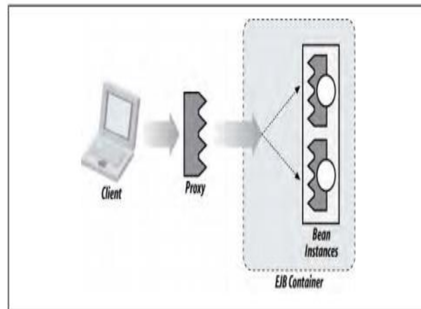


Figure 2-1. Client invoking upon a proxy object, responsible for delegating the call along to the EJB Container

Figure 2-1. C
Container

2. What is session bean? Explain the types of session beans.

Session Beans If EJB is a grammar, session beans are the verbs. Session beans contain business methods.

Types of Session Bean

There are 3 types of session bean.

1) Stateless Session Bean: It doesn't maintain state of a client between multiple method calls.

2) Stateful Session Bean: It maintains state of a client across multiple requests.
3) Singleton Session Bean: One instance per application, it is shared between clients and supports concurrent access.

Stateless session beans (SLSBs) Stateless session beans are useful for functions in which state does not need to be carried from invocation to invocation. The Container will often create and destroy instances. This allows the Container to hold a much smaller number of objects in service, hence keeping• memory footprint down.

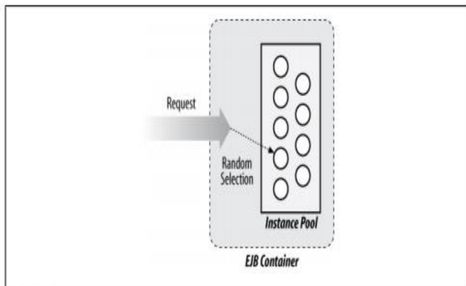


Figure 2-2. An SLSB Instance Selector picking an instance at random

Stateful session beans (SFSBs) Stateful session beans differ from SLSBs in that every request upon a given proxy reference is guaranteed to ultimately invoke upon the same bean instance. SFSB invocations share conversational state. Each SFSB proxy object has an isolated session context, so calls to one session will not affect another. Stateful sessions, and their corresponding bean instances, are created sometime before the first invocation upon a proxy is made to its target instance (Figure 2-3). They live until the client invokes a method that the bean provider has marked as a remove event, or until the Container decides to remove the session.

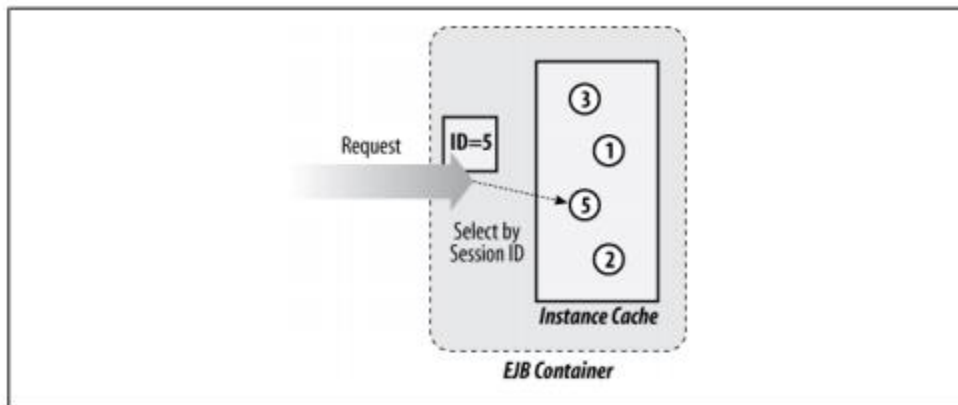


Figure 2-3. Stateful session bean creating and using the correct client instance, which lives inside the EJB Container, to carry out the invocation

Singleton beans Sometimes we don't need any more than one backing instance for our business objects. All requests upon a singleton are destined for the same bean instance, The Container doesn't have much work to do in choosing the target (Figure 2-4). The singleton session bean may be marked to eagerly load when an application is deployed; therefore, it may

be leveraged to fire application lifecycle events. This draws a relationship where deploying a singleton bean implicitly leads to the invocation of its lifecycle callbacks. We'll put this to good use when we discuss singleton beans.

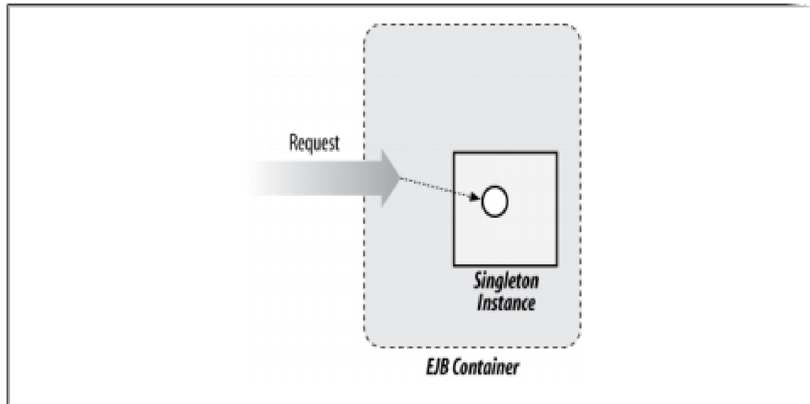
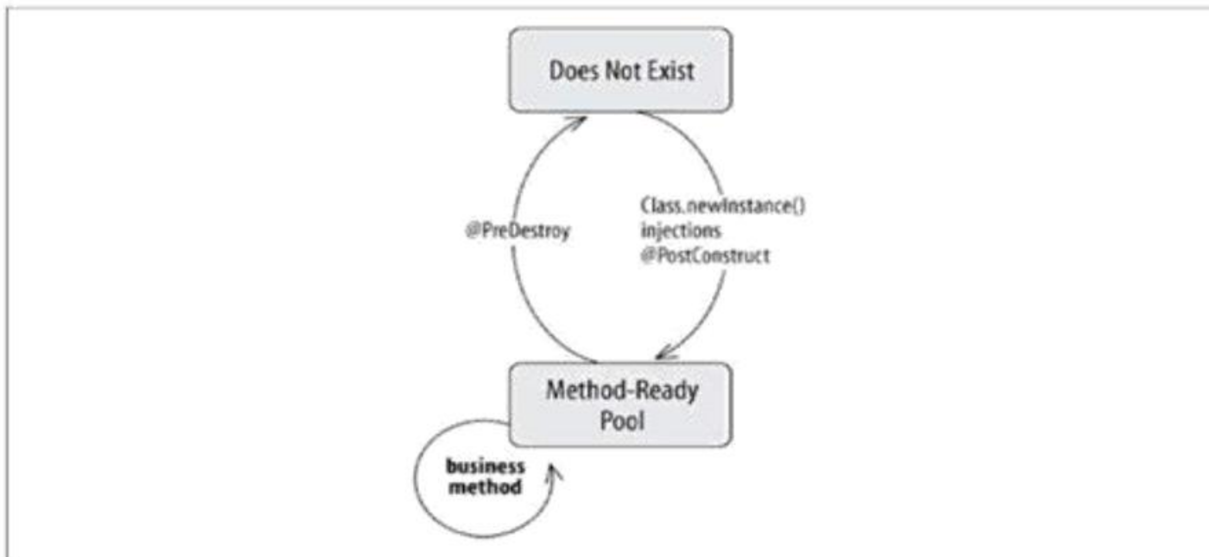


Figure 2-4. Conceptual diagram of a singleton session bean with only one backing bean instance

3. Explain about the Statelful session bean with its life cycle

- It has only two states: Does Not Exist and Method-Ready Pool.
- The Method-Ready Pool is an instance pool of stateless session bean



The Does Not Exist State

When a bean 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

- Stateless bean instances enter the Method-Ready Pool as the container needs them.
- When the EJB server is first started, it may create a number of stateless bean instances and enter them into the Method-Ready Pool.
- When the number of stateless instances servicing client requests 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.

1. First, the bean instance is instantiated by invoking the Class.newInstance() method on the stateless bean class.
2. Second, the container injects any resources that the bean's metadata has requested via an injection annotation or XML deployment descriptor.
3. **Finally, the EJB container will fire a post-construction event.**

The bean class can register for this event by annotating a method with @javax.annotation.PostConstruct.

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

- **Once an instance is in the Method-Ready Pool,** it is ready to service client requests.
- When a client invokes a business method on an EJB object, the method call is delegated to any available instance in the Method-Ready Pool.
- While the instance is executing the request, it is unavailable for use by other EJB objects.
- Once the instance has finished, it is immediately available to any EJB object that needs it.

- Stateless session instances are dedicated to an EJB object only for the duration of a single method call.

- **When an instance is swapped in,** its SessionContext changes to reflect the context of the EJB object and the client invoking the method.

- Once the instance has finished servicing the client, it is disassociated from the EJB object and returned to the Method-Ready Pool.

- **Clients that need a remote or local reference** to a stateless session bean begin by having the reference injected or by looking up the stateless bean in JNDI.

- The reference returned does not cause a session bean instance to be created or pulled from the pool until a method is invoked on it.

- PostConstruct is invoked only once in the lifecycle of an instance: when it is transitioning from the Does Not Exist state to the Method-Ready Pool.

Transitioning out of the Method-Ready Pool: The death of a stateless bean 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 when a PreDestroy event on the bean is triggered. The bean class can register for this event by annotating a method with @javax.annotation.PreDestroy.

- The container calls this annotated method when the PreDestroy event is fired. This callback method can be of any name, but it must return void, have no parameters, and throw no checked exceptions.

4. What is message driven bean? Discuss the life cycle of the MDB with neat diagram

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

MDB asynchronously receives the message and processes it. A message driven bean receives message from queue or topic, so you must have the knowledge of JMS API.

- It has only two states: Does Not Exist and Method-Ready Pool.

- The Method-Ready Pool is an instance pool of stateless session bean

The Does Not Exist State

When a bean 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

- Stateless bean instances enter the Method-Ready Pool as the container needs them.

- When the EJB server is first started, it may create a number of stateless bean instances and enter them into the Method-Ready Pool.

- When the number of stateless instances servicing client requests 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.

5. First, the bean instance is instantiated by invoking the Class.newInstance() method on the stateless bean class.

6. Second, the container injects any resources that the bean's metadata has requested via an injection annotation or XML deployment descriptor.

7. **Finally, the EJB container will fire a post-construction event.**

The bean class can register for this event by annotating a method with @javax.annotation.PostConstruct.

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

- **Once an instance is in the Method-Ready Pool,** it is ready to service client requests.

- When a client invokes a business method on an EJB object, the method call is delegated to any available instance in the Method-Ready Pool.

- While the instance is executing the request, it is unavailable for use by other EJB objects.

- Once the instance has finished, it is immediately available to any EJB object that needs it.

5. Summarise the Entity Relationship

- There are four types of cardinality:
  - *one-to-one,*
  - *one-to-many,*
  - *many-to-one,*
  - *many-to-many.*
- *E*ach relationship can be either
  - *Unidirectional*
  - *bidirectional.*

1. One-to-one unidirectional
- The relationship between an employee and an address.

2. One-to-one bidirectional
- The relationship between an employee and a computer(computer ID for servicing)
- Given an employee, we'll need to be able to look up the computer ID for tracing purposes.

3. One-to-many unidirectional
- The relationship between an employee and a phone number.
- An employee can have many phone numbers (business, home, cell, etc.).

4. One-to-many bidirectional
- The relationship between an employee (manager) and direct reports.
- Given a manager, we'd like to know who's working under him or her.
- able to find the manager for a given employee

5. Many-to-one unidirectional
- The relationship between a customer and his or her primary employee contact.

6. Many-to-many unidirectional
- The relationship between employees and tasks to be completed.
- Each task may be assigned to a number of employees, and employees may be responsible for many tasks.
- Given a task we need to find its related employees, but not the other way around.

7. Many-to-many bidirectional
- The relationship between an employee and the teams to which he or she belongs.
- Teams may also have many employees, and we'd like to do lookups in both directions.

6.Explain the steps in JDBC process. Give an example

The following 5 steps are the basic steps involve in connecting a Java application with Database using JDBC.
Register the Driver
Create a Connection
Create SQL Statement
Execute SQL Statement
Closing the connection
Register the Driver
`Class.forName()` is used to load the driver class explicitly.
Example to register with JDBC-ODBC Driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Create a Connection
`getConnection()` method of **DriverManager** class is used to create a connection.
Syntax
getConnection(String url)
getConnection(String url, String username, String password)
getConnection(String url, Properties info)
Example establish connection with Oracle Driver
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:XE","username","password");

Create SQL Statement
`createStatement()` method is invoked on current **Connection** object to create a SQL Statement.
Syntax
public Statement createStatement() throws SQLException
Example to create a SQL statement
Statement s=con.createStatement();

Execute SQL Statement
`executeQuery()` method of **Statement** interface is used to execute SQL statements.
Syntax
public ResultSet executeQuery(String query) throws SQLException
Example to execute a SQL statement

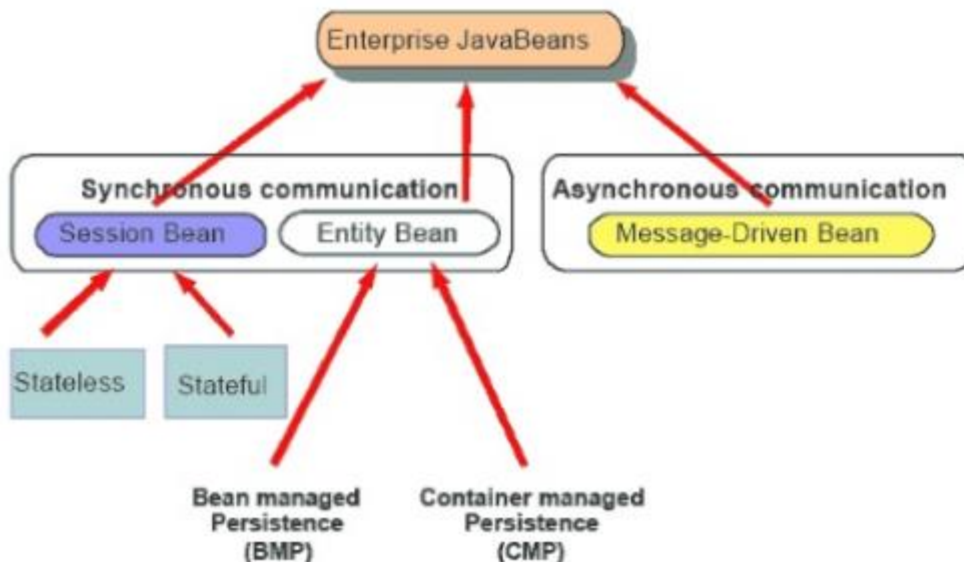
```
ResultSet rs=s.executeQuery("select * from user");
  while(rs.next())
  {
   System.out.println(rs.getString(1)+" "+rs.getString(2));
  }
```

Closing the connection
After executing SQL statement you need to close the connection and release the session.
The `close()` method of **Connection** interface is used to close the connection.
Syntax
public void close() throws SQLException

Example of closing a connection

```
con.close();
import java.sql.*;
class OracleCon{
public static void main(String args[]){
try{
//step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");
  //step2 create  the connection object
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
//step3 create the statement object
Statement stmt=con.createStatement();
//step4 execute query
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
//step5 close the connection object
con.close();
  }catch(Exception e){ System.out.println(e);}
}
}
```

7. Discuss the classes of EJB and depict tfhe various components of interaction with a neat diagram.



Session Beans If EJB is a grammar, session beans are the verbs. Session beans contain business methods.

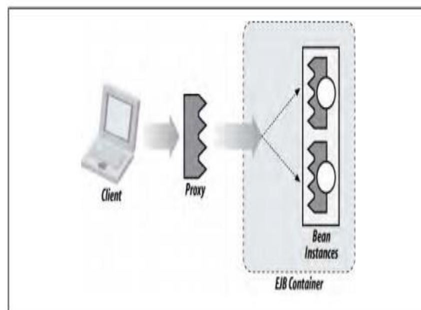


Figure 2-1. Client invoking upon a proxy object, responsible for delegating the call along to the EJB Container

Types of Session Bean

There are 3 types of session bean.

1) Stateless Session Bean: It doesn't maintain state of a client between multiple method calls.

2) Stateful Session Bean: It maintains state of a client across multiple requests.

3) Singleton Session Bean: One instance per application, it is shared between clients and supports concurrent access.

Stateless session beans (SLSBs) Stateless session beans are useful for functions in which state does not need to be carried from invocation to invocation. The Container will often create and destroy instances. This allows the Container to hold a much smaller number of objects in service, hence keeping• memory footprint down.

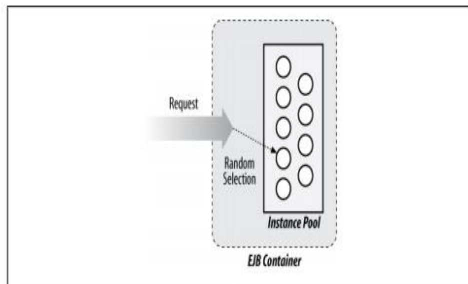


Figure 2-2. An SLSB Instance Selector picking an instance at random

Stateful session beans (SFSBs) Stateful session beans differ from SLSBs in that every request upon a given proxy reference is guaranteed to ultimately invoke upon the same bean instance. SFSB invocations share conversational state. Each SFSB proxy object has an isolated session context, so calls to one session will not affect another. Stateful sessions, and their corresponding bean instances, are created sometime before the first invocation upon a proxy is made to its target instance (Figure 2-3). They live until the client invokes a method that the bean provider has marked as a remove event, or until the Container decides to remove the session.

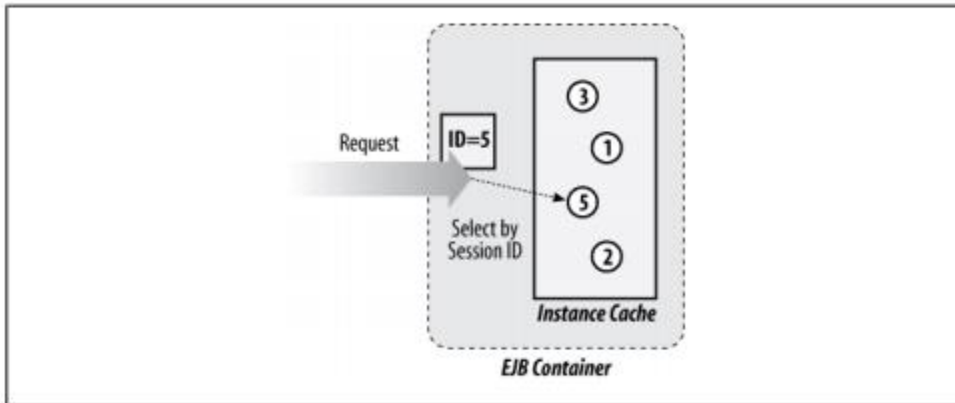


Figure 2-3. Stateful session bean creating and using the correct client instance, which lives inside the EJB Container, to carry out the invocation

Singleton beans Sometimes we don't need any more than one backing instance for our business objects. All requests upon a singleton are destined for the same bean instance, The Container doesn't have much work to do in choosing the target (Figure 2-4). The singleton session bean may be marked to eagerly load when an application is deployed; therefore, it may be leveraged to fire application lifecycle events. This draws a relationship where deploying a singleton bean implicitly leads to the invocation of its lifecycle callbacks. We'll put this to good use when we discuss singleton beans.

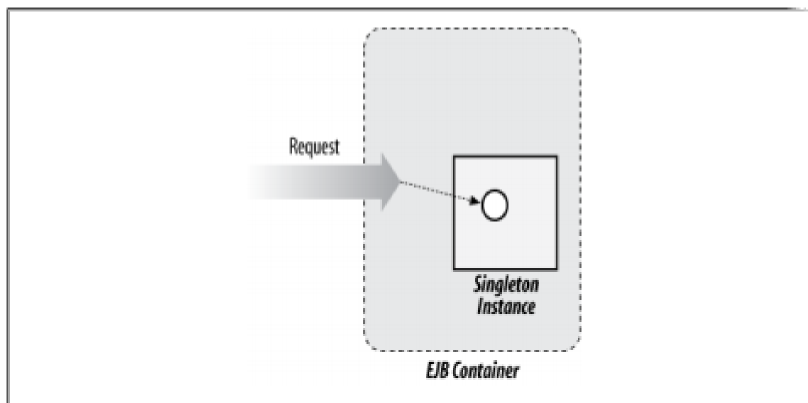


Figure 2-4. Conceptual diagram of a singleton session bean with only one backing bean instance

MDB

Asynchronous messaging is a paradigm in which two or more applications communicate via a message describing a business event. EJB 3.1 interacts with messaging systems via the Java Connector Architecture (JCA) 1.6 (http://jcp.org/en/jsr/detail?id=322), which acts as an abstraction layer that enables any system to be adapted as a valid sender. The message-driven bean, in turn, is a listener that consumes messages and may either handle them directly or delegate further processing to other EJB components. The asynchronous characteristic of this exchange means that a message sender is not waiting for a response, so no return to the caller is provided

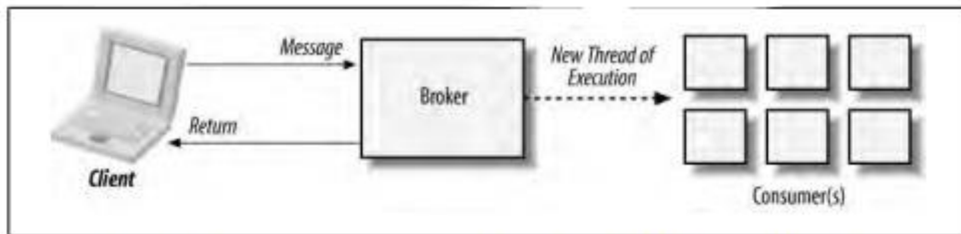


Figure 2-5. Asynchronous invocation of a message-driven bean, which acts as a listener for incoming events

Entity Beans While session beans are our verbs, entity beans are the nouns. Their aim is to express an object view of resources stored within a Relational Database Management System (RDBMS)—a process commonly known as object-relational mapping. Like session beans, the entity type is modeled as a POJO, and becomes a managed object only when associated with a construct called the javax.persistence.EntityManager, a container-supplied service that tracks state changes and synchronizes with the database as necessary. A client who alters the state of an entity bean may expect any altered fields to be propagated to persistent storage. Frequently the EntityManager will cache both reads and writes to transparently streamline performance, and may enlist with the current transaction to flush state to persistent storage automatically upon invocation completion.

Unlike session beans and MDBs, entity beans are not themselves a server-side component type. Instead, they are a view that may be detached from management and used just like any stateful object. When detached (disassociated from the EntityManager), there is no database association, but the object may later be re-enlisted with the EntityManager such that its state may again be synchronized. Just as session beans are EJBs only within the context of the Container, entity beans are managed only when registered with the EntityManager. In all other cases entity beans act as POJOs, making them extremely versatile.

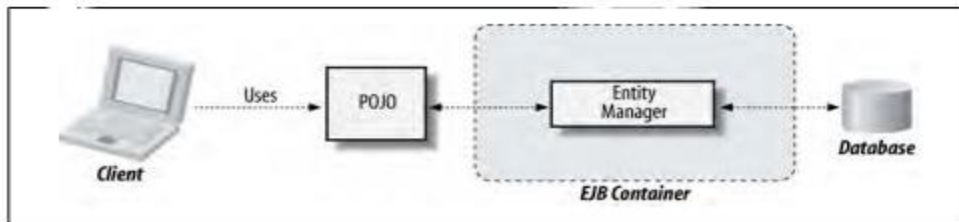


Figure 2-6. Using an EntityManager to map between POJO object state and a persistent relational database

8) a

Write the short note about the following

(i) Persistence Context

(ii) XML Deployment Descriptor

(i) Persistence Context

- **A persistence context is a set of managed entity object instances**.
- Persistence contexts are managed by an entity manager.
- The entity manager tracks all entity objects within a persistence context for changes and updates made, and flushes these changes to the database using the flush mode rules
- Once a persistence context is closed, all managed entity object instances become detached and are no longer managed.
- Once an object is detached from a persistence context, it can no longer be managed by an entity manager, and any state changes to this object instance will not be synchronized with the database.
- When a persistence context is closed, all managed entity objects become detached and are unmanaged.
- There are two types of persistence contexts:
    - transaction-scoped persistence context
    - extended persistence context.

Transaction Scoped Persistence context

- Everything executed
- Either fully succeed or fully fail

(ii) XML Deployment Descriptor

A deployment descriptor describes how EJBs are managed at runtime and enables the customization of EJB behavior without modification to the EJB code.
A deployment descriptor is written in a file using XML syntax.
Add file is packed in the Java Archive (JAR) file along with the other files that are required to deploy the EJB. It includes classes and component interfaces that are necessary for each EJB in the package.
An EJB container references the deployment descriptor file to understand how to deploy and manage EJBs contained in package.
 The deployment descriptor identifies the types of EJBs that are contained in the package as well as other attributes, such as how transactions are managed.

8. Write an EJB program that demonstrates session bean with proper business logic. 10

Source Code :

Calculator.java

```
package package1;
import javax.ejb.Stateless;
```

```
@Stateless
public class Calculator implements CalculatorLocal
{
   @Override
   public Integer Addition(int a, int b) {
      return a+b;
   }

   @Override
   public Integer Subtract(int a, int b) {
      return a-b;
   }

   @Override
   public Integer Multiply(int a, int b) {
      return a*b;
   }
   @Override
    public Integer Division(int a, int b) {
      return a/b;
   }
}
```

CalculatorLocal.java

```
package package1;
import javax.ejb.Local;
public interface CalculatorLocal {
      Integer Addition(int a, int b);
      Integer Subtract(int a, int b);
      Integer Multiply(int a, int b);
      Integer Division(int a, int b);
}
```

Servlet1.java

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import package1.CalculatorLocal;

public class Servlet1 extends HttpServlet {
   @EJB
   private CalculatorLocal calculator;
   protected void processRequest(HttpServletRequest request, HttpServletResponse response)
```

```
        throws ServletException, IOException
    {
        response.setContentType("text/html;charset=UTF-8");
      try (PrintWriter out = response.getWriter())
      {
        out.println("Output :  "+ "<br/>");
        int a;
        a = Integer.parseInt(request.getParameter("num1"));
        int b;
        b=Integer.parseInt(request.getParameter("num2"));
        out.println("Number1 :  " + a + "<br/>");
        out.println("Number2 :  " + b+ "<br/>");
        out.println("Addition  : " + calculator.Addition(a, b)+ "<br/>");
        out.println("Subtraction  :" + calculator.Subtract(a, b)+ "<br/>");
        out.println("Multiplication  :"+calculator.Multiply(a, b)+ "<br/>");
        out.println("Division  :"+calculator.Division(a, b)+ "<br/>");

      }
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
      processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
      processRequest(request, response);
    }

    @Override
    public String getServletInfo()
    {
      return "Short description";
    }
}
```

index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Calculator</title>
```

```
    </head>
    <body>
        <form method="get" action="Servlet1">
            Enter Number1 :  <input type="text" name="num1"/><br/>
            Enter Number2 :  <input type="text" name="num2"/><br/>
            <input type="submit" value="Submit"/>
        </form>
    </body>
</html>
```

10 What is entity bean Explain the JAVA persistence model in detail

```
package Persist;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String usnno;
    private String name;
    private int mark;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }

    @Override
```

```
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Student)) {
            return false;
        }
        Student other = (Student) object;
        if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }
    @Override
    public String toString() {
        return "Persist.Student[ id=" + id + " ]";
    }
    public String getUsnno() {
        return usnno;
    }

    public void setUsnno(String usnno) {
        this.usnno = usnno;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    public int getMark() {
        return mark;
    }

    public void setMark(int mark) {
        this.mark = mark;
    }

}
```

StudServlet.java

```
package WebClient;
import Persist.Student;
import Persist.StudentFacadeLocal;
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class StudServlet extends HttpServlet
{
    @EJB
    private StudentFacadeLocal studentFacade;
      protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        Student obj=new Student();
        obj.setUsnno(request.getParameter("usnno"));
        obj.setName(request.getParameter("name"));
        obj.setMark(Integer.parseInt(request.getParameter("mark")));
        studentFacade.create(obj);

        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Student Data</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Congrats : Student " + request.getParameter("name") + " Record is
created successfully </h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
    @Override
    public String getServletInfo() {
        return "Short description";
    }// </editor-fold>
}
```

Index.html

```
<html>
    <head>
```

```
        <title>TODO supply a title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
         <form method="get" action="StudServlet">
        Enter USN No. :<input type ="text" name ="usnno"/><br/>
        Enter Name    :<input type ="text" name ="name"/><br/>
        Enter Mark    :<input type ="text" name ="mark"/><br/>
            <input type="submit" value="Submit"/>
    </form>
    </body>
</html>
```