CMR
INSTITUTE OF
USN
TECHNOLOGY

Internal Assessment Test 3–May 2018

Sub:	Advanced Java Programming							Code:	16MCA 41
Date:	21.05.2018	*Duration:*	*90 mins*	*Max Marks:*	*50*	*Sem:*	*4*	*Branch:*	*MCA*

1. a) Explain about Stateful Session Bean.

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 . 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

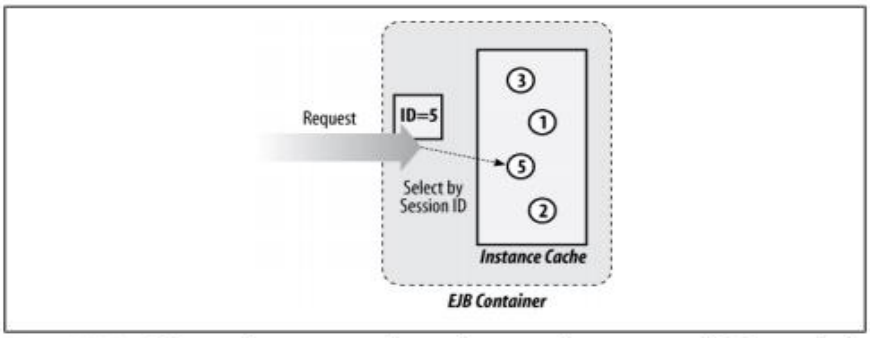


Figure 2-3. Stateful session bean creating and using the correct client instance, which lives inside the EJB Container, to carry out the invocation

The biggest difference between the stateful session bean and the other bean types is that stateful session beans do not use instance pooling. Stateful session beans are dedicated to one client for their entire lives, so swapping or pooling of instances isn't possible.† When they are idle, stateful session bean instances are simply evicted from memory.

Life Cycle:

The lifecycle of a stateful session bean has three states: Does Not Exist, Method-Ready, and Passivated.

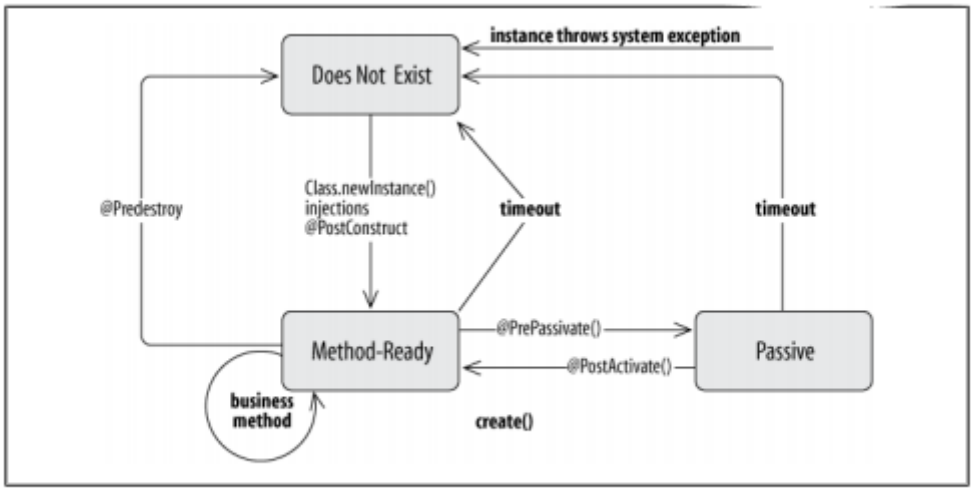


Figure 6-2. Stateful session bean lifecycle

The Does Not Exist State A stateful bean instance in the Does Not Exist state has not been instantiated yet. It doesn't exist in the system's memory.

The Method-Ready State The Method-Ready state is the state in which the bean instance can service requests from its clients. This section explores the instance's transition into and out of the Method-Ready state.

Transitioning into the Method-Ready state When a client invokes the first method on the stateful session bean reference, the bean's lifecycle begins. The container invokes newInstance() on the bean class, creating a new instance of the bean. Next, the container injects any dependencies into the bean instance. At this point, the bean instance is assigned to the client referencing it. Finally, just like stateless session beans, the container invokes any @PostConstruct callbacks if there is a method in the bean class that has this annotation applied. Once @PostConstruct has completed, the container continues with the actual method call.

Life in the Method-Ready state While in the Method-Ready state, the bean instance is free to receive method invocations from the client, which may involve controlling the taskflow of other beans or accessing the database directly. During this time, the bean can maintain conversational state and open resources in its instance variables.

Transitioning out of the Method-Ready state Bean instances leave the Method-Ready state to enter either the Passivated state or the Does Not Exist state. Depending on how the client uses the stateful bean, the EJB container's load, and the passivation algorithm used by the vendor, a bean instance may be passivated (and activated) several times in its life, or not at all. If the bean is removed, it enters the Does Not Exist state. A client application can remove a bean by invoking a business interface method annotated as @Remove.

The container can also move the bean instance from the Method-Ready state to the Does Not Exist state if the bean times out. Timeouts are declared at deployment time in a vendor-specific manner. When a timeout occurs in the Method-Ready state, the container may, but is not required to, call any @PreDestroy callback methods. A stateful bean cannot time out while a transaction is in progress.

The Passivated State During the lifetime of a stateful session bean, there may be periods of inactivity when the bean instance is not servicing methods from the client. To conserve resources, the container can passivate the bean instance by preserving its conversational state and evicting the bean instance from memory.

b) Write a short note on Entity Relationship and its types.

In order to model real-world business concepts, entity beans must be capable of forming relationships. For instance, an employee may have an address; we'd like to form an association between the two in our

database model. The address could be queried and cached like any other entity, yet a close relationship would be forged with the Employee entity.
Seven types of relationships can exist between entity beans. There are four types of cardinality: one-to-one, one-to-many, many-to-one, and many-to-many. In addition, each relationship can be either unidirectional or bidirectional.
One-to-one unidirectional The relationship between an employee and an address. You clearly want to be able to look up an employee's address, but you probably don't care about looking up an address's employee.
One-to-one bidirectional The relationship between an employee and a computer. Given an employee, we'll need to be able to look up the computer ID for tracing purposes. Assuming the computer is in the tech department for servicing, it's also helpful to locate the employee when all work is completed.
One-to-many unidirectional The relationship between an employee and a phone number. An employee can have many phone numbers (business, home, cell, etc.). You might need to look up an employee's phone number, but you probably wouldn't use one of those numbers to look up the employee.
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. Similarly, we'd like to be able to find the manager for a given employee. (Note that a many-to-one bidirectional relationship is just another perspective on the same concept.)
Many-to-one unidirectional The relationship between a customer and his or her primary employee contact. Given a customer, we'd like to know who's in charge of handling the account. It might be less useful to look up all the accounts a specific employee is fronting, although if you want this capability you can implement a many-to-one bidirectional relationship.
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. For now we'll assume that given a task we need to find its related employees, but not the other way around. (If you think you need to do so, implement it as a bidirectional relationship.)
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.

2. What are the container services provided by component model of EJB. Describe them in detail.

In the case of EJB, the Component Model defines our interchangeable parts, and the Container Services are specialists that perform work upon them. The Specification provides:

1. Dependency injection
2. Concurrency
3. Instance pooling/caching
4. Transactions - *Transaction management*—Declarative transactions provide a means for the developer to easily delegate the creation and control of transactions to the container.
5. Security - declarative security provides a means for the developer to easily delegate the enforcement of security to the container.
6. Timers
7. Naming and object stores - The EJB container and server will provide the EJB with access to naming services. These services are used by local and remote clients to look up the EJB and by the EJB itself to look up resources it may need.
8. Interoperability
9. Lifecycle callbacks
10. Interceptors
11. Java Enterprise Platform integration

1.Dependency Injection (DI)

- A component based approach to software design brings with it the complication of inter-module

communication.

- Tightly coupling discrete units together violates module independence and separation of concerns,
- While using common look up code leads to the maintenance of more plumbing
- As EJB is a component-centric architecture, it provides a means to reference dependent modules in decoupled fashion.
- The container provide the implementation at deployment time.

In pseudocode, this looks like:

```
prototype UserModule
{
  // Instance Member
  @DependentModule
  MailModule mail;

  // A function
  function mailUser()
  {
    mail.sendMail("me@ejb.somedomain");
  }
}
```

The @DependentModule annotation serves two purposes:
• Defines a dependency upon some service of type MailModule. UserModule may not deploy until this dependency is satisfied.
• Marks the instance member mail as a candidate for injection. The container will populate this field during deployment.

2. Concurrency

- **Assuming each Service is represeznted by one instance,** dependency injection alone is a fine solution for a single-threaded application; only one client may be accessing a resource at a given time.
- However, this quickly becomes a problem in situations where a centralized server is fit to serve many simultaneous requests.
- Deadlocks, livelocks, and race conditions are some of the possible nightmares arising out of an environment in which threads may compete for shared resources.
- These are hard to anticipate, harder to debug, and are prone to first exposing themselves in production!
- EJB allows the application developer to sidestep the problem entirely thanks to a series of concurrency policies.

3. 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 (see Figure 3-1).

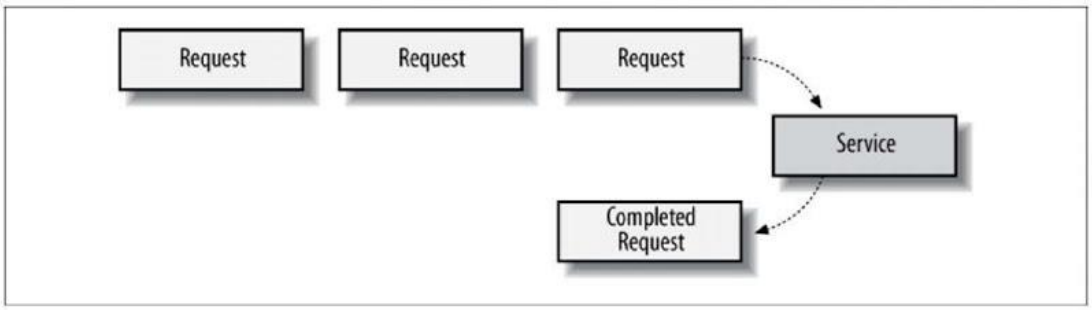


Figure 3-1. Client requests queuing for service

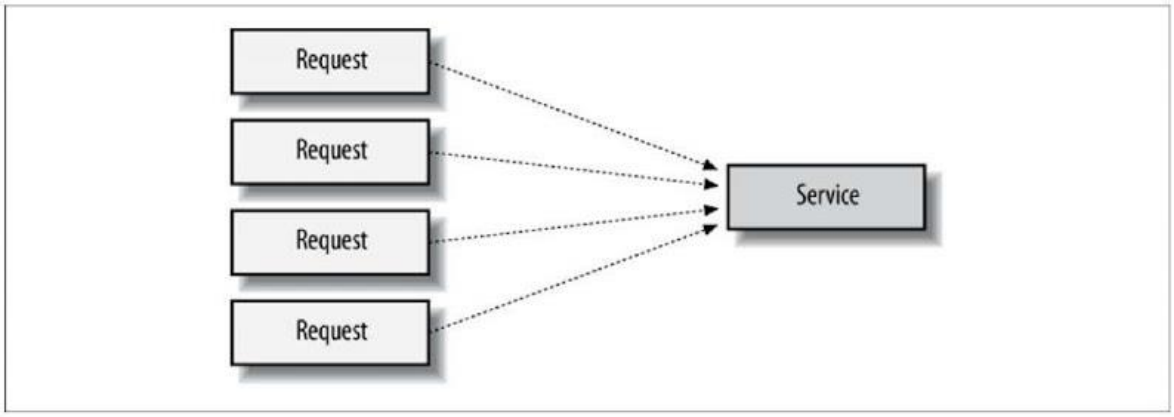


Figure 3-2. Many invocations executing concurrently with no queuing policy

EJB addresses this problem through a technique called instance pooling, in which each module is allocated some number of instances with which to serve incoming requests (Figure 3-3).
Many vendors provide configuration options to allocate pool sizes appropriate to the work being performed, providing the compromise needed to achieve optimal throughput.

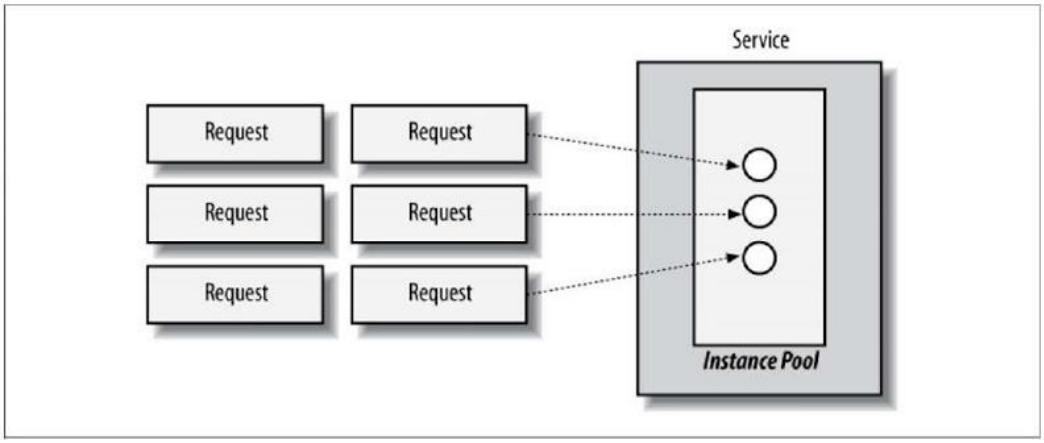


Figure 3-3. A hybrid approach using a pool

4. 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.

5. 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.

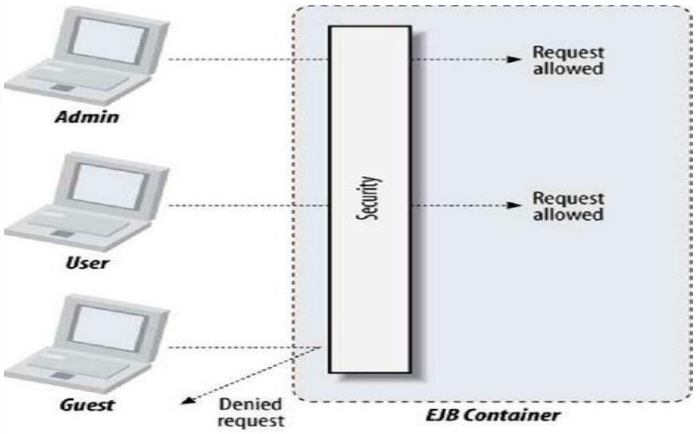


Figure 15-1. EJB security permitting access based upon the caller's role

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

6.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.

7. Naming and Object Stores

- They provide clients with a mechanism for locating distributed objects or resources.
- To accomplish this, a naming service must fulfill two requirements:
- Object Binding - Object binding is the association of a distributed object with a natural language name or identifier
- Lookup API - A lookup API provides the client with an interface to the naming system; it simply allows us to connect with a distributed service and request a remote reference to a specific object.
- A Enterprise JavaBeans mandates the use of Java Naming and Directory Interface as lookup API on Java clients.
- JNDI supports just about any kind of naming and directory service.
- Java client applications can use JNDI to initiate a connection to an EJB server and locate a specific EJB.

8. Interoperability

- Our application may want to consume data from or provide services to other programs, perhaps written in different implementation languages.
- There are a variety of open standards that address this inter-process communication, and EJB leverages these.
- Interoperability is a vital part of EJB.
- The specification includes the required support for Java RMI-IIOP for remote method invocation and provides

for transaction, naming, and security interoperability.

- EJB also requires support for JAX-WS, JAX-RPC, Web Services for Java EE, and Web Services Metadata for the Java Platform specifications

9.Lifecycle Callbacks

- **Some services require some initialization** or cleanup to be used properly.
- For example, a file transfer module may want to open a connection to a remote server before processing requests to transfer files and should safely release all resources before being brought out of service.
- For component types that have a lifecycle, EJB allows for callback notifications, which act as a hook for the bean provider to receive these events.

In the case of our file transfer bean, this may look like:

```
prototype FileTransferService
{

   @StartLifecycleCallback
   function openConnection(){ ... }

   @StopLifecycleCallback
   function closeConnection() { ... }
}
```

Here we've annotated functions to open and close connections as callbacks; they'll be invoked by the container as their corresponding lifecycle states are reached.

10.Interceptors

- EJB provides aspectised handling of many of the container services the specification cannot possibly identify all cross cutting concerns facing your project
- EJB makes to possible to define custom inceptors upon business methods and lifecycle callbacks
- Example : We want to measure the execution time of all invocations to a particular method
  We would write an interceptor

```
prototype MetricsInterceptor
{
        Function intercept(Invocation invocation)
        {
                Time startTime = getTime();
                Invocation.continue();
                Time endTime=getTime();
                log("Took : "+ (endTime-startTime));
        }
}
\
```

Then we could apply this to methods as we would like

```
@ApplyInterceptor(MetricsInterceptor.class)
Function myLoginMethod{……}

@ApplyInterceptor(MetricsInterceptor.class)
Function myLogoutMethod{……}
```

11.Platform Integration

As a key technology within the Java Enterprise Edition (JEE) 6, EJB aggregates many of the other platform

frameworks and APIs:

- Java Transaction Service
- Java Persistence API
- Java Naming and Directory Interface (JNDI)
- Security Services
- Web Services

- **In most cases,** the EJB metadata used to define this integration is a simplified view of the underlying services.
- This gives bean providers a set of powerful constructs right out of the box, without need for additional configuration.
- EJB is also one of the target models recognized by the new Java Contexts and Dependency Injection specification**.**
- This ads a unified binding to both Java Enterprise and Standard editions across many different component types.

3. Write a short note about

a)Implementation class

The contract, implemented as interfaces in Java, defines what our service will do, and leaves it up to the implementation classes to decide how it's done. Remember that the same interface cannot be used for both @Local and @Remote, so we'll make some common base that may be extended.

```
public interface CalculatorCommonBusiness {
 /** * Adds all arguments * *
 @return The sum of all arguments */
int add(int... arguments);
 }
public class CalculatorBeanBase implements CalculatorCommonBusiness {
 /** *
 {
@link CalculatorCommonBusiness#
add(int...)
}
 */ @Override
 public int add(final int... arguments)
{ // Initialize int result = 0;
// Add all arguments for (final int arg : arguments)
{
result += arg;
}
// Return return result;
}
 }
```

This contains the required implementation of CalculatorCommonBusiness. add(int...). The bean implementation class therefore has very little work to do.

```
 import javax.ejb.LocalBean;
 import javax.ejb.Stateless;
@Stateless
@LocalBean
public class SimpleCalculatorBean extends CalculatorBeanBase
 {
/* * Implementation supplied by common base class */
}
```

The function of our bean implementation class here is to bring everything together and define the EJB metadata. Compilation will embed two important bits into the resultant .class file. First, we have an SLSB, as noted by the @Stateless annotation. And second, we're exposing a no-interface view

b) Integration Testing of EJB

There are three steps involved in performing integration testing upon an EJB.
First, we must package the sources and any descriptors into a standard Java Archive
Next, the resultant deployable must be placed into the container according to a vendor-specific mechanism.
Finally, we need a standalone client to obtain the proxy references from the Container and invoke upon them.

Packaging

A standard jar tool that can be used to assemble classes, resources, and other metadata into a unified JAR file, which will both compress and encapsulate its contents.

Deployment into the Container The EJB Specification intentionally leaves the issue of deployment up to the vendor's discretion.

The client

Instead of creating POJOs via the new operator, we'll look up true EJB references via JNDI. JNDI is a simple store from which we may request objects keyed to some known address.

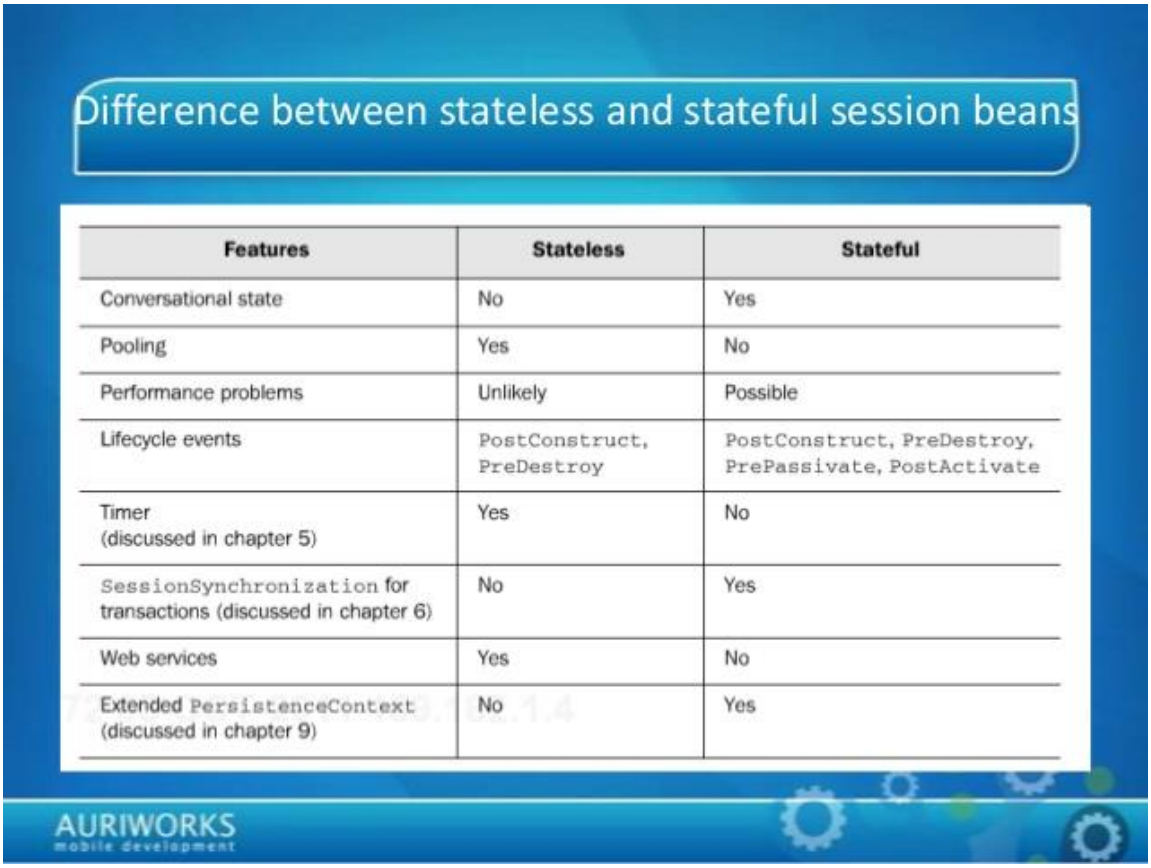
4. List the differences between stateful and stateless session bean

Stateless:

1) Stateless session bean maintains across method and transaction
2) The EJB server transparently reuses instances of the Bean to service different clients at the per-method level (access to the session bean is serialized and is 1 client per session bean per method.
3) Used mainly to provide a pool of beans to handle frequent but brief reuests. The EJB server transparently reuses instances of the bean to service different clients.
4) Do not retain client information from one method invocation to the next. So many requir the client tomaintain on the client side which can mean more complex client code.
5) Client passes needed information as parameters to the business methods.6) Performance can be improved due to fewer connections across the network.

Stateful:

1) A stateful session bean holds the client session's state.
2) A stateful session bean is an extesion of the client that creates it.
3) Its fields contain a conversational state on behalf of the session object's client. This state describes the conversation represented by a specific client/session object pair.
4) Its lifetime is controlled by the client.
5) Cannot be shared between clients.



Features	Stateless	Stateful
Conversational state	No	Yes
Pooling	Yes	No
Performance problems	Unlikely	Possible
Lifecycle events	`PostConstruct`, `PreDestroy`	`PostConstruct`, `PreDestroy`, `PrePassivate`, `PostActivate`
Timer (discussed in chapter 5)	Yes	No
`SessionSynchronization` for transactions (discussed in chapter 6)	No	Yes
Web services	Yes	No
Extended `PersistenceContext` (discussed in chapter 9)	No	Yes

5. a) Define java bean and state its advantages. Explain the features of java bean

JAVA BEAN

- JavaBeans is a
  - portable,
  - platform-independent component model
  - written in the Java programming language.
- The JavaBeans architecture was built
  - through a collaborative industry effort and
  - enables developers to write reusable components in the Java programming language.
- Java Bean components are known as beans.
- Beans are dynamic in that they can be changed or customized.

Advantage of Java Bean

- A Bean obtains all the benefits of Java's **"write-once, run-anywhere"** paradigm.
- The properties, events, and methods of a Bean that are exposed to an **application builder tool can be controlled**.
- A Bean may be designed to operate correctly in different locales, which makes it **useful in global markets**.
- Auxiliary software can be provided to help a person **configure a Bean**. This software is only needed when the design-time parameters for that component are being set. It does

not need to be included in the run-time environment.

- The **configuration settings** of a Bean can be saved in **persistent storage** and restored at a later time.
- A Bean may **register to receive events** from other objects and can generate events that are sent to other objects.

Bean Key Concept

- Builder tools discover a Bean's features (that is, its properties, methods, and events) by a process known as *introspection(examine own thoughts)*. Beans support introspection in two ways:
  - **By adhering to specific rules, known as *design patterns*,** when naming Bean features
  - **By explicitly providing property, method, and event information with a related *Bean Information* class**
- *Properties* are a Bean's appearance and behavior characteristics that can be changed at design time.
  - Builder tools introspect on a Bean to discover its properties, and expose those properties for manipulation.
- Beans expose properties so they can be *customized* at design time. Customization is supported in two ways:
  - By using property editors,
  - By using more sophisticated Bean customizers.
- Beans use *events* to communicate with other Beans.
  - A Bean that wants to receive events (a listener Bean) registers its interest with the Bean that fires the event (a source Bean).
  - Builder tools can examine a Bean and determine which events that Bean can fire (send) and which it can handle (receive).
- *Persistence* enables Beans to save and restore their state.
  - Once you've changed a Beans properties, you can save the state of the Bean and restore that Bean at a later time, property changes intact.
  - JavaBeans uses Java Object Serialization to support persistence.
- A Bean's *methods* are no different than Java methods, and can be called from other Beans or a scripting environment. By default all public methods are exported.
- Beans are designed to be understood by builder tools
  - all key APIs, including support for events, properties, and persistence, have been designed to be easily read and understood by human programmers as well.

5. b) Define java bean and state its advantages. Explain the features of java bean

This bean will just draw itself in red, and when you click it, it will display a count of the number of time it has been clicked. We can place this bean in the BDK's demo directory, so we create a directory named bean and store the class files for this bean in that directory.

```
package sunw.demo.bean;
import java.awt.*;
import java.awt.event.*;
public class bean extends Canvas
```

```
{
    int count;
    public bean()
    {          addMouseListener (new MouseAdapter()
               {   public void mousePressed(MouseEvent me)
                   {
                         clicked();
                    }
               });
               count = 0;
               setSize(200,00);
   }
public void clicked()
{
    count++;
    repaint();
}
public void paint(Graphics g)
{
   g.setColor(Color.RED);
   g.fillRect(0,0,20,30);
   g.drawString(—Click Count= —+count,50,50);
}
}
```

6.a. How are jar files created and used? Explain its advantages.

The basic format of the command for creating a JAR file is:

jar cf jar-file input-file(s)

The options and arguments used in this command are:

The c option indicates that you want to create a JAR file.

The f option indicates that you want the output to go to a file rather than to stdout.

jar-file is the name that you want the resulting JAR file to have. You can use any filename for a JAR file. By convention, JAR filenames are given a .jar extension, though this is not required.

The input-file(s) argument is a space-separated list of one or more files that you want to include in your JAR file. The input-file(s) argument can contain the wildcard * symbol. If any of the "input-files" are directories, the contents of those directories are added to the JAR archive recursively.

The c and f options can appear in either order, but there must not be any space between them.

The jar tool provides many switches, some of them are as follows:

-c creates new archive file

-v generates verbose output. It displays the included or extracted resource on the standard output.

-m includes manifest information from the given mf file.

-f specifies the archive file name

-x extracts files from the archive file

Advantages:

Security: You can digitally sign the contents of a JAR file.

Decreased download time: for Applets and Java Web Start
Compression: efficient storage
Packaging for extensions: extend JVM (example Java3D)
Package Sealing: enforce version consistency
all classes defined in a package must be found in the same JAR file
Package Versioning: hold data like like vendor and version information
Portability: the mechanism for handling JAR files is a standard part of the Java platform's core API

b. What is a manifest file? Mention its importance.

The manifest is a special file that can contain information about the files packaged in a JAR file. By tailoring this "meta" information that the manifest contains, you enable the JAR file to serve a variety of purposes.

applications Bundled as JAR Files: If an application is bundled in a JAR file, the Java Virtual Machine needs to be told what the entry point to the application is. An entry point is any class with a public static void main(String[] args) method. This information is provided in the Main-Class header, which has the general form:

Main-Class: classname

The value classname is to be replaced with the application's entry point.

Download Extensions: Download extensions are JAR files that are referenced by the manifest files of other JAR files. In a typical situation, an applet will be bundled in a JAR file whose manifest references a JAR file (or several JAR files) that will serve as an extension for the purposes of that applet. Extensions may reference each other in the same way. Download extensions are specified in the Class-Path header field in the manifest file of an applet, application, or another extension. A Class-Path header might look like this, for example:

Class-Path: servlet.jar infobus.jar acme/beans.jar

With this header, the classes in the files servlet.jar, infobus.jar, and acme/beans.jar will serve as extensions for purposes of the applet or application. The URLs in the Class-Path header are given relative to the URL of the JAR file of the applet or application.

Package Sealing: A package within a JAR file can be optionally sealed, which means that all classes defined in that package must be archived in the same JAR file. A package might be sealed to ensure version consistency among the classes in your software or as a security measure. To seal a package, a Name header needs to be added for the package, followed by a Sealed header, similar to this:

Name: myCompany/myPackage/
Sealed: true

The Name header's value is the package's relative pathname. Note that it ends with a '/' to distinguish it from a filename. Any headers following a Name header, without any intervening blank lines, apply to the file or package specified in the Name header. In the above example, because the Sealed header occurs after the Name: myCompany/myPackage header, with no blank lines between, the Sealed header will be interpreted as applying (only) to the package myCompany/myPackage.

Package Versioning: The Package Versioning specification defines several manifest headers to hold versioning information. One set of such headers can be assigned to each package. The

versioning headers should appear directly beneath the Name header for the package. This example shows all the versioning headers:

Name: java/util/
Specification-Title: "Java Utility Classes"
Specification-Version: "1.2"
Specification-Vendor: "Sun Microsystems, Inc.".
Implementation-Title: "java.util"
Implementation-Version: "build57"
Implementation-Vendor: "Sun Microsystems, Inc."

7.a. Explain any four annotations used in EJB along with their meaning.

	Name	Description
	javax.ejb.Stateless	Specifies that a given EJB class is a stateless session bean. Attributes name − Used to specify name of the session bean. mappedName − Used to specify the JNDI name of the session bean. description − Used to provide description of the session bean.
	javax.ejb.Stateful	Specifies that a given EJB class is a stateful session bean. Attributes name − Used to specify name of the session bean. mappedName − Used to specify the JNDI name of the session bean. description − Used to provide description of the session bean.
	javax.ejb.Message DrivenBean	Specifies that a given EJB class is a message driven bean. Attributes name − Used to specify name of the message driven bean. messageListenerInterface− Used to specify message listener interface for the message driven bean. activationConfig − Used to specify the configuration

		details of the message-driven bean in an operational environment of the message driven bean. mappedName − Used to specify the JNDI name of the session bean. description − Used to provide description of the session bean.
	javax.ejb.EJB	Used to specify or inject a dependency as EJB instance into another EJB. Attributes name − Used to specify name, which will be used to locate the referenced bean in the environment. beanInterface − Used to specify the interface type of the referenced bean. beanName − Used to provide name of the referenced bean. mappedName − Used to specify the JNDI name of the referenced bean. description − Used to provide description of the referenced bean.

b) Write a short note on Singleton bean

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.

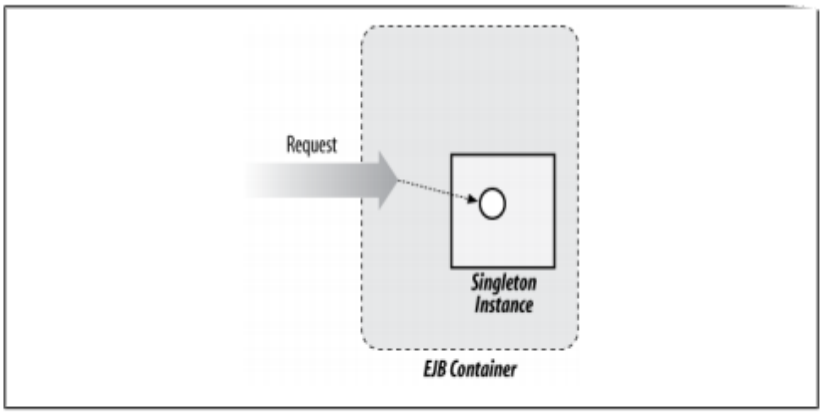


Figure 2-4. Conceptual diagram of a singleton session bean with only one backing bean instance

LifeCycle:

The life of a singleton bean is very similar to that of the stateless session bean; it is either not yet instantiated or ready to service requests. In general, it is up to the Container to determine when to

create the underlying bean instance, though this must be available before the first invocation is executed. Once made, the singleton bean instance lives in memory for the life of the application and is shared among all requests.

- It has only two states: Does Not Exist and Method-Ready Pool.
- The Method-Ready Pool is an instance pool of session bean objects that are not in use.

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

- Session bean instances enter the Method-Ready Pool as the container needs them.
- When the EJB server is first started, it may create a bean instance and enter them into the Method-Ready 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 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 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.

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 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 transi-tioning from the Does Not Exist state to the Method-Ready Pool.

Tansitioning out of the Method-Ready Pool: The death of a session 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.

8.a) Explain the lifecycle of MDB.

The MDB instance's lifecycle has two states: Does Not Exist and Method-Ready Pool. The Method-Ready Pool is similar to the instance pool used for stateless session beans.§ Figure 8-5 illustrates the states and transitions that an MDB instance goes through in its lifetime

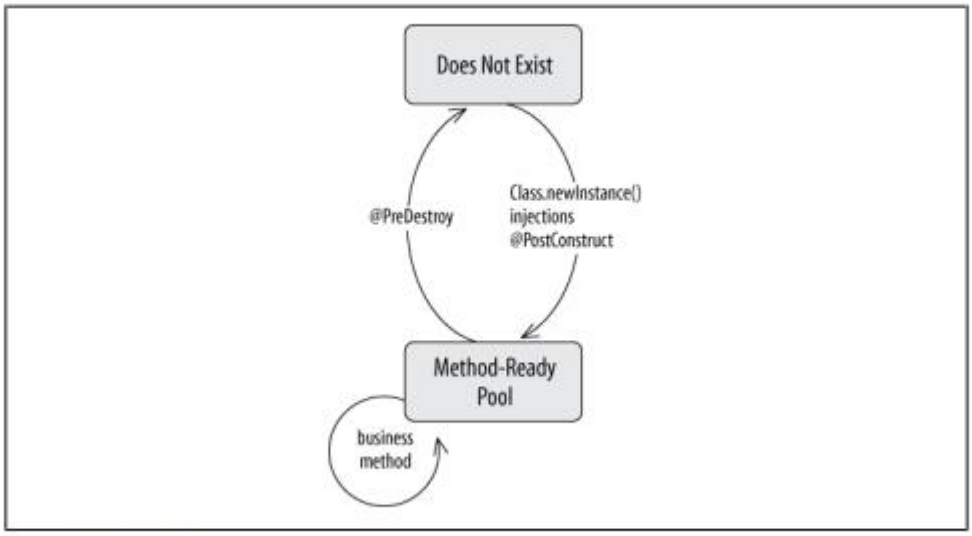


Figure 8-5. MDB lifecycle

The Does Not Exist State When an MDB instance is in the Does Not Exist state, it is not an instance in the memory of the system. In other words, it has not been instantiated yet.
The Method-Ready Pool MDB instances enter the Method-Ready Pool as the container needs them. When the EJB server is first started, it may create a number of MDB instances and enter them into the Method-Ready Pool (the actual behavior of the server depends on the implementation). When the number of MDB instances handling incoming messages is insufficient, more can be created and added to the pool.
Transitioning to the Method-Ready Pool When an instance transitions from the Does Not Exist state to the Method-Ready Pool, three operations are performed on it. First, the bean instance is instantiated by invoking the Class.newInstance() method on the bean implementation class. Second, the container injects any resources that the bean's metadata has requested via an injection annotation or XML deployment descriptor.
Finally, the EJB container will invoke the PostConstruct callback if there is one. The bean class may or may not have a method that is annotated with @javax.ejb.PostConstruct. If it is present, the container will call this annotated method after the bean is instantiated. This @PostConstruct annotated method can be of any name and visibility, but it must return void, have no parameters, and throw no checked exceptions. The bean class may define only one @PostConstruct method (but it is not required to do so). @MessageDriven public class MyBean implements MessageListener { @PostConstruct public void myInit() {} MDBs are not subject to activation, so they can maintain open connections to resources for their entire lifecycles.‖ The @PreDestroy

method should close any open resources before the stateless session bean is evicted from memory at the end of its lifecycle.

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

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

b) What are the types of JMS Messaging models used? Explain with neat diagram.

JMS Messaging Models JMS provides two types of messaging models: publish-and-subscribe and point-topoint. The JMS specification refers to these as messaging domains. In JMS terminology, publish-and-subscribe and point-to-point are frequently shortened to pub/sub and p2p (or PTP), respectively. In the simplest sense, publish-and-subscribe is intended for a one-to-many broadcast of messages. Point-to-point, on the other hand, is intended for a message that is to be processed once

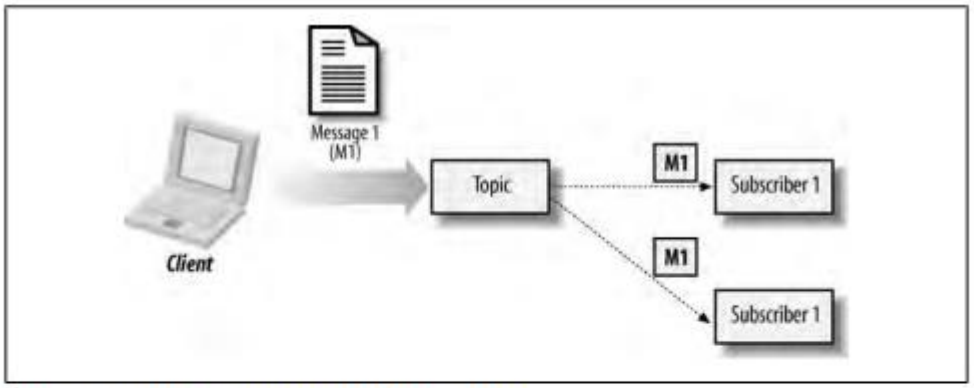


Figure 8-3. Publish-and-subscribe model of messaging

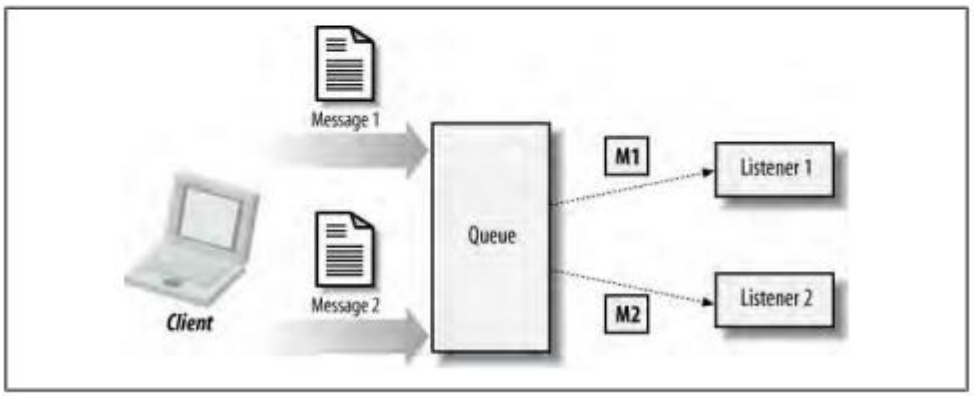


Figure 8-4. Point-to-point model of messaging

Each messaging domain (i.e., pub/sub and p2p) has its own set of interfaces and classes for sending and receiving messages. This results in two different APIs, which share some common types. JMS 1.1 introduced a Unified API that allows developers to use a single set of interfaces and classes for both messaging domains.

Publish-and-subscribe In publish-and-subscribe messaging, one producer can send a message to many consumers through a virtual channel called a topic. Consumers can choose to subscribe to a topic. Any messages addressed to a topic are delivered to all the topic's consumers. The pub/sub messaging model is largely a push-based model, in which messages are automatically broadcast to consumers without the consumers having to request or poll the topic for new messages.

In this pub/sub messaging model, the producer sending the message is not dependent on the consumers receiving the message. JMS clients that use pub/sub can establish durable subscriptions that allow consumers to disconnect and later reconnect and collect messages that were published while they were disconnected.

Point-to-point The point-to-point messaging model allows JMS clients to send and receive messages both synchronously and asynchronously via virtual channels known as queues. The p2p messaging model has traditionally been a pull- or polling-based model, in which messages are requested from the queue instead of being pushed to the client automatically.* A queue may have multiple receivers, but only one receiver may receive each message. As shown earlier, the JMS provider takes care of doling out the messages among JMS clients, ensuring that each message is processed by only one consumer. The JMS specification does not dictate the rules for distributing messages among multiple receivers.

9 What is entity bean Explain the JAVA persistence model in detail.

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.

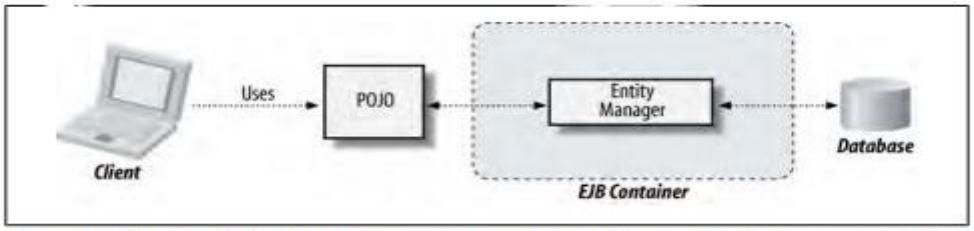


Figure 2-6. Using an EntityManager to map between POJO object state and a persistent relational database

Persistance Model:

Persistence provides an ease-of-use abstraction on top of JDBC so that your code may be isolated from the database and vendor-specific peculiarities and optimizations. It can also be described as an object-to-relational mapping engine (ORM), which means that the Java Persistence API can automatically map your Java objects to and from a relational database. In addition to object mappings, this service also provides a query language that is very SQL-like but is tailored to work with Java objects rather than a relational schema. In short, JPA handles the plumbing between Java and SQL. EJB provides convenient integration with JPA via the entity bean.

Entity beans gain powerful services when used within the context of the container. In the case of persistence, Entity instances may become managed objects under the control of a service called the javax.persistence.EntityManager

Entities Are POJOs

Entities, in the Java Persistence specification, are plain old Java objects (POJOs). You allocate them with the new() operator just as you would any other plain Java object. Their state is not synchronized with persistent storage unless associated with an EntityManager. For instance, let's look at a simple example of an Employee entity:

```
import javax.persistence.Entity;
import javax.persistence.Id;
/**
 * Represents an Employee in the system. Modeled as a simple
 * value object with some additional EJB and JPA annotations.
 *
 * @author <a href="mailto:andrew.rubinger@jboss.org">ALR</a>
 * @version $Revision: $
 */
@Entity
// Mark that we're an Entity Bean, EJB's integration point
// with Java Persistence
public class Employee
{
 /**
 * Primary key of this entity
 */
 @Id
 // Mark that this field is the primary key
 private Long id;
 /**
 * Name of the employee
 */
 private String name;
/**
 * Default constructor, required by JPA
 */
 public Employee()
 {
 }
 /**
 * Convenience constructor
 */
```

```
public Employee(final long id, final String name)
{
// Set
this.id = id;
this.name = name;
}
/**
* @return the id
*/
public Long getId()
{
return id;
}
/**
* @param id the id to set
*/
public void setId(final Long id)
{
this.id = id;
}
/**
* @return the name
*/
public String getName()
{
return name;
}
/**
* @param name the name to set
*/
public void setName(final String name)
{
this.name = name;
}
/**
* {@inheritDoc}
* @see java.lang.Object#toString()
*/
@Override
public String toString()
{
return "Employee [id=" + id + ", name=" + name + "]";
}
}
```

If we allocate instances of this Employee class, no magic happens when new() is invoked. Calling the new operator does not magically interact with some underlying service to create the Employee in the database:

```
// This is just an object
Employee hero = new Employee(1L,"Trey Anastasio");
```

Allocated instances of the Customer class remain POJOs until you ask

the Entity Manager to create the entity in the database.

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 with in a persistence context for changes and updates made, and flushes these changes to the database using the flush mode rules discussed later in this chapter. 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.

10. Explain bound and constrained properties of design patterns.

Bound Properties:

A bean that has a bound property generates an event when the property is changed.

Bound Properties are the properties of a JavaBean that inform its listeners about changes in its values.

Bound Properties are implemented using the **PropertyChangeSupport** class and its methods.

Bound Properties are always registered with an external event listener.

The event is of type ***PropertyChangeEvent*** and is sent to objects that previously egistered an interest in
receiving such notifications

bean with bound property - Event source

Bean implementing listener -- event target

In order to provide this notification service a Java Bean needs to have the following two methods:

```
public void addPropertyChangeListener(PropertyChangeListener p) {
changes.addPropertyChangeListener(p);
}
public void
}
removePropertyChangeListener(PropertyChangeListener p) {
changes.removePropertyChangeListener(p);
```

PropertyChangeListener is an interface declared in the java.beans package. Observers which want to be notified
of property changes have to implement this interface, which consists of only one method:

```
public interface PropertyChangeListener extends
EventListener { public void
propertyChange(PropertyChangeEvent e );

}
```

Constrained Properties:

It generates an event when an attempt is made to change it value

Constrained Properties are implemented using the ***PropertyChangeEvent*** class.

The event is sent to objects that previously registered an interest in receiving an such notification

Those other objects have the ability to veto the proposed change

This allows a bean to operate differently according to the runtime environment

A bean property for which a change to the property results in validation by another bean. The other bean may
reject the change if it is not appropriate.

Constrained Properties are the properties that are protected from being changed by other JavaBeans. Constrained
Properties are registered with an external event listener that has the ability to either accept or reject the change in
the value of a constrained property.

Constrained Properties can be retrieved using the get method. The prototype of the get method is:

Syntax:

public string get<ConstrainedPropertyName>()

Can be specified using the set method. The prototype of the set method is:

Syntax :public string set<ConstrainedPropertyName>(String str)throws
PropertyVetoException