UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES
CAMPUS ARAGÓN

Enterprise JavaBeans como una alternativa confiable para el desarrollo de sistemas distribuidos

T E S I S

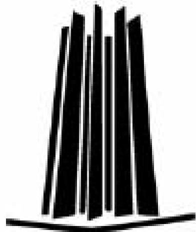
QUE PARA OBTENER EL TÍTULO DE:
INGENIERO EN COMPUTACIÓN

P R E S E N T A:

Yescas Quiroz Juan Carlos

ASESOR DE TESIS:
Ing. Jesús Hernández Cabrera

MÉXICO, 2007.



Universidad Nacional
Autónoma de México

Biblioteca Central

Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Esta tesis esta dedicada a Paulina, tías y tíos

Contenido

Introducción 5
Objetivos 6

1 **Introducción a J2EE** 7

1.1 Introducción 7
1.2 Arquitectura J2EE 7
1.2.1 Características de la Arquitectura J2EE 10
1.2.2 Contenedores de la plataforma J2EE 12
1.3 Tecnologías y servicios empleados para desarrollar y soportar aplicaciones J2EE 12
1.3.1 Tecnologías J2EE 12
1.3.2 Servicios proporcionados por J2EE 14
1.4 Roles en J2EE 15
1.5 Resumen 16

2 **Introducción a los Enterprise JavaBeans** 17
2.1 Introducción 17
2.2 Arquitecturas de objetos distribuidas 17
2.3 Arquitectura EJB 19
2.3.1 El contenedor y sus requerimientos 19
2.3.2 Componentes EJB 20
2.3.3 JavaBeans y Enterprise JavaBeans 21
2.4 Roles específicos para EJB 21
2.5 Desarrollo de un enterprise bean 22
2.6 Resumen 28

3 **Beans de Sesión (Session Beans)** 29
3.1 Introducción 29
3.2 Beans de Sesión 29
3.2.2 Component interface 31
3.2.3 SessionBean interface 32
3.3 Stateful Session Beans 33
3.3.1 Desarrollo de un Stateful Session Bean 33
3.3.2 Arquitectura de un Stateful Session Bean 37
3.3.3 Ciclo de vida de un Sateful Session Bean 38
3.4 Stateless Session Beans 39
3.4.1 Desarrollo de un Stateless Session Bean 39
3.4.2 Arquitectura de un Stateless Session Bean 42
3.4.3 Ciclo de vida de un Stateless Session Bean 43

3.5 Resumen 44

4 **Beans de Entidad (Entity Beans) 45**
4.1 Introducción 45
4.2 Beans de entidad 45
4.2.1 Características de un bean de entidad 46
4.2.2 Home interface 46
4.2.3 Component interface 47
4.2.4 EntityBean interface 48
4.3 Desarrollo de un Entity Bean 49
4.4 Resumen 54

5 **Beans para el Manejo de Mensajes (Message-driven beans) 55**
5.1 Introducción 55
5.2 Comunicación asíncrona con Beans para el manejo de mensajes 55
5.2.1 Funcionamiento de los beans para el manejo de mensajes 56
5.2.2 Desarrollo de un Message Driven Bean 57
5.3 Ciclo de vida de un MDB 60
5.4 Tipos de destinación 61
5.4.1 Destinación de tipo Queue 61
5.4.2 Destinación de tipo Topic 62
5.5 Resumen 62

6 **Despliegue de Enterprise Beans usando el servidor de aplicaciones JBoss 63**
6.1 Introducción 63
6.2 Instalación de JBoss y Ant 63
6.2.1 Estructura de directorios de JBoss 65
6.3 Despliegue del bean de sesión "Quote" 66
6.4 Despliegue del bean de sesión "Beer" 68
6.5 Despliegue del bean de sesión "Discount" 71
6.6 Despliegue del bean de entidad "Consumidor" 73
6.7 Despliegue del bean para el manejo de mensajes "Notificar" 76

Conclusión 79

Bibliografía 80

Introducción

La arquitectura J2EE esta construida en base a componentes que en conjunto permiten a los desarrolladores construir aplicaciones J2EE robustas que aprovechan las eficiencias de la tecnología distribuida.

J2EE es una arquitectura de Multicapa, cada una de sus capas desempeña una función específica en la aplicación, lo cual permite cambiar fácilmente el funcionamiento de cada capa, sin impactar a las demás capas, como suceden en los sistemas de una o dos capas.

La arquitectura J2EE tiene cuatro capas, las cuales son presentadas a continuación:

- Capa Cliente
- Capa Web
- Capa de Negocios
- Capa de Datos

La capa cliente es la encargada de enviar las peticiones de un cliente, tal como un navegador o una aplicación, a la capa de negocios o a la capa web, para que sea procesada la petición.

La capa web se encarga de procesar peticiones HTTP, las cuales solicitan determinadas tareas, esta puede interactuar directamente con la capa de datos, o con la capa de negocios.

La capa de negocios, la cual juega un papel importante en nuestro estudio, es la encargada de contener toda la lógica de negocio de nuestra aplicación, y se encarga de interactuar con la capa de datos, proporcionándonos un soporte transaccional para nuestra aplicación.

En la capa de datos, es la que contiene todos los datos requeridos por nuestra aplicación, para que sean transformados en información. En esta capa por lo general está conformada de sistemas manejadores de base de datos.

De todas las capas anteriores, la que es objeto de nuestro estudio, es la capa de negocios, la cual contiene a los Enterprise Beans. Los enterprise beans pueden representar procesos de la aplicación, o entidades de nuestra base de datos, dependiendo el tipo de enterprise bean que se utilice. Existen principalmente tres tipos de Enterprise Beans: Session Beans, Entity Bean y Message-Driven Beans, los cuales tienen determinadas reglas para su construcción.

Para desplegar a los enterprise beans, es necesario contar con un servidor de aplicaciones, tal como weblogic, Websphere Application Server o JBoss. En nuestro estudio, se utilizara JBoss, y se verá alguna de sus ventajas, entre las cuales se encuentra su arquitectura microkernel.

Objetivos

Objetivo General:

Desarrollo de Enterprise Beans para aplicaciones distribuidas, usando JBoss como servidor de aplicaciones.

Objetivo Particular

- Conocer que es un Enterprise Bean y el lugar que ocupan en una arquitectura Multicapa.
- Aprender los tres tipos de Enterprise Beans y sus diferencias.
- Desarrollo e implementación de Session Beans.
- Desarrollo e implementación de Entity Beans.
- Desarrollo e implementación de Message-Driven Beans.
- Despliegue de Enterprise Beans en el servidor de aplicaciones JBoss.

Introducción a J2EE

1.1 Introducción

Al pasar de los años, la tecnología Java ha crecido y evolucionado, dejando atrás la idea de las aplicaciones basadas en applets, para desarrollar aplicaciones robustas que trabajen el lado del servidor. Los paquetes inicialmente introducidos por el J2SDK (Java Software Development Kit) para la construcción de aplicaciones basadas en hilos, soporte para I/O y protocolos de red han continuado mejorando, lo cual desemboca en la construcción de aplicaciones a gran escala, como servidores Web, servidores de aplicaciones, etc. Todo lo anterior condujo a la creación de la edición empresarial para la plataforma Java, comúnmente conocida como Java 2 Enterprise Edición (J2EE).

Java 2 Enterprise Edition (J2EE) es una plataforma basada en Java que ofrece un ambiente multicapa para el desarrollo, despliegue y ejecución de aplicaciones empresariales.

La adopción de esta plataforma ha sido notable. Un ingrediente crítico en el desarrollo de J2EE es el entorno de colaboración fomentado por Sun Mycrosystems, en el cual los proveedores y técnicos se reúnen en Java Community Program (JCP) para crear e implementar tecnologías basadas en Java.

Para entender como esta conformada la plataforma J2EE, la analizaremos en tres caminos:

- Arquitectura J2EE
- Tecnologías y servicios empleados para desarrollar y soportar aplicaciones J2EE
- Basada en los roles que existen para construir aplicaciones J2EE reales

1.2 Arquitectura J2EE

Una arquitectura multicapa es aquella que esta formada por varias capas, donde podemos definir ***capa*** como un grupo de tecnologías que proporcionan uno o más servicios a sus clientes.

Una arquitectura multicapa se compone de clientes, recursos, componentes y contenedores. Un ***cliente*** se refiere a un programa que solicita un servicio de un componente. Un ***recurso*** es cualquier cosa que necesita el componente para proporcionar el servicio, mientras que un ***componente*** es parte de una capa que consiste de un conjunto de clases o un programa que realiza una función para proporcionar un servicio. Un ***contenedor*** es el software que gestiona al componente.

La relación entre un componente y un contenedor en ocasiones se conoce como contrato, cuyos términos se rigen por una interfaz de programación de aplicaciones (API). Una API define las reglas que debe seguir un componente y los servicios que recibe el componente del contenedor.

Un contenedor se encarga de la persistencia, la gestión de recursos, la seguridad, los hilos y otros servicios a nivel sistema para los componentes asociados con él. Los componentes son responsables de implementar la lógica de negocio. Esto significa que los programadores se pueden concentrar en codificar las reglas de negocio en los componentes sin tener que preocuparse sobre los servicios de sistema de bajo nivel, ni gestionar los servicios de red.

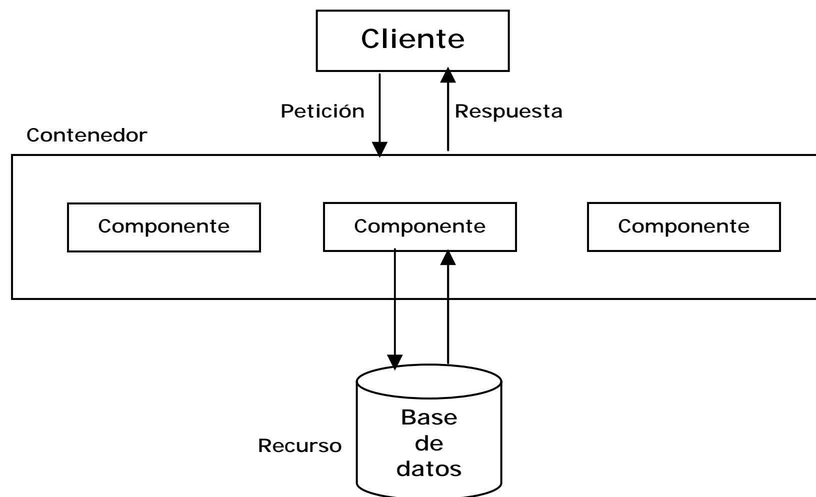


Figura 1.1 Arquitectura Multicapa, compuesta de un cliente, contenedor y recursos.

J2EE es una arquitectura de cuatro capas. Estas capas son la capa cliente (también conocida como capa de presentación o de aplicación), la capa web, la capa de Enterprise JavaBeans (también conocida como la capa de negocio) y la capa de sistemas de información empresarial (EIS, Enterprise Information System). Cada capa se enfoca en proporcionar a una aplicación un tipo específico de funcionalidad.

Un punto importante a destacar es que algunas APIs no están asociadas solo a una capa en particular, por ejemplo, la API para procesar XML se puede utilizar en la capa web y de negocios, mientras que otras APIs, como la API para Enterprise JavaBeans, solo están asociadas a una capa determinada.

La capa cliente está compuesta por los programas que interactúan con el usuario. Estos programas piden datos al usuario y convierten su respuesta en peticiones que se reenvían a un componente que procesa la petición y devuelve el resultado al programa cliente. El componente puede operar en cualquier capa, aunque la mayoría de las peticiones de clientes van dirigidas a componentes de la capa web.

La capa web proporciona a la aplicación J2EE las APIs necesarias para interactuar con el World Wide Web. Los componentes de esta capa utilizan HTTP para recibir peticiones de y enviar respuestas a clientes que pueden estar en cualquier capa.

Un componente de la capa web recibe una petición de datos de un cliente y la pasa a la capa de Enterprise JavaBeans (EJB), en la que un EJB interactúa con el DBMS para atender la petición. Una petición se envía a un Enterprise JavaBean mediante la API de invocación remota de métodos (RMI). El EJB recibe los datos del DBMS y los envía a la capa web, la cual los reenvía a la capa cliente, donde se muestra al usuario.

La capa de Enterprise JavaBeans (EJB) contiene la lógica de negocio de las aplicaciones J2EE. Es aquí donde se encuentran uno o más Enterprise JavaBeans, cada uno de los cuales contiene reglas de negocio, que llaman los clientes en forma directa. En la capa de negocios, los EJB llaman a componentes y recursos como el DBMS en la capa de sistemas de información empresarial (Enterprise Information Systems o EIS).

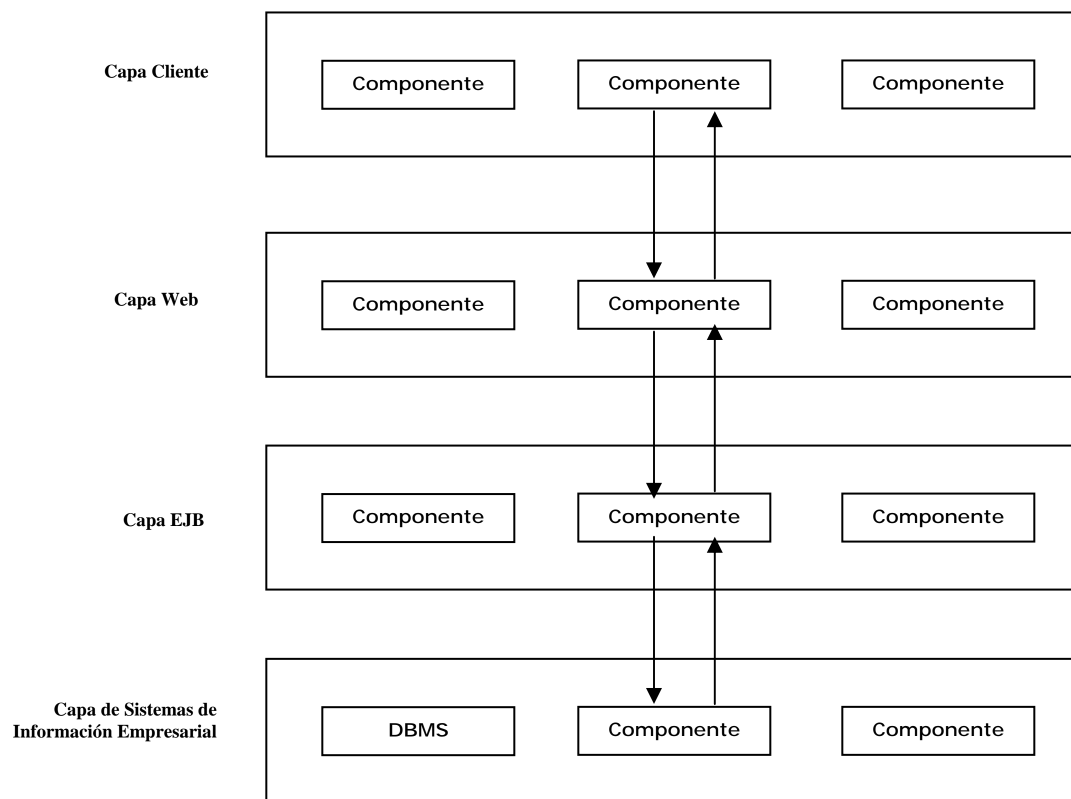


Figura 1.2 **Arquitectura J2EE formada de 4 capas, la capa del cliente, capa web, capa de EJB, y la capa de Sistemas de Información Empresarial.**

En la figura anterior podemos notar que cada petición pasa de una capa a otra antes de ser atendida, pero esto no siempre es cierto, ya que una aplicación que está en la capa cliente puede solicitar un servicio directamente de la capa de negocios.

A continuación se muestra un cliente accediendo directamente a la capa de negocios:

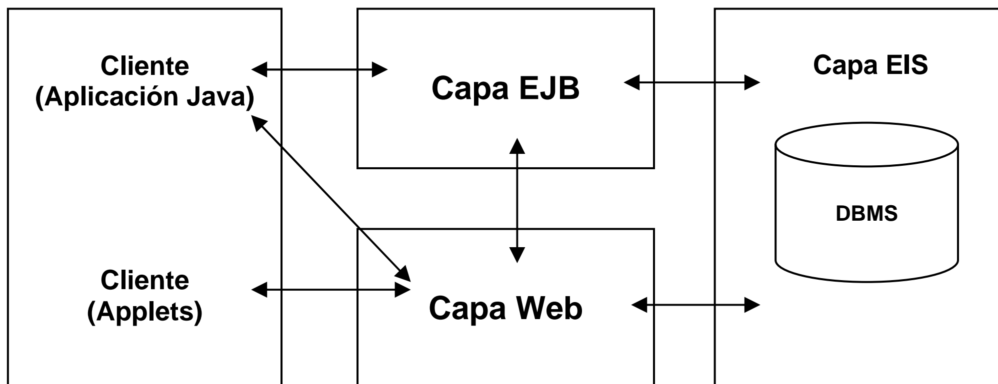


Figura 1.3 **Interacción entre capas, donde un cliente puede acceder directamente a la capa EJB o la capa Web, y donde la capa de EIS es accedida ya sea desde la capa EJB o capa Web.**

1.2.1 Características de la Arquitectura J2EE

Para poder comprender porque la arquitectura J2EE es una tecnología robusta, tenemos que describir sus características, en las cuales nos enfocamos a continuación.

Arquitectura Multicapa

Una capa es un grupo de componentes de software que ofrecen un específico tipo de funcionalidad o servicio. J2EE esta basada en un arquitectura Multicapa, esto permite a los componentes de software estar distribuidos en diferentes máquinas, lo que facilita la escalabilidad, seguridad y división de responsabilidad durante el desarrollo, despliegue y ejecución. Dependiendo de la aplicación, un cliente puede acceder a la capa EJB directamente o solo a través de la capa web. Un elemento en la capa web puede acceder a la base directamente o solo a través de la capa EJB, la decisión final depende del diseño de la aplicación. Esta flexibilidad en la arquitectura deja varias opciones para la construcción de aplicaciones dependiendo del tamaño de la aplicación o complejidad de la misma. La capa web es también llamada capa de presentación porque su función es manejar las entradas del usuario y presentar los resultados al mismo. Similarmente, la capa EJB es algunas veces llamada la capa de negocios porque es ahí donde la lógica de negocio de la aplicación es implementada. La capa de datos es llamada capa Enterprise Information System (EIS).

Ambiente Distribuido

El ambiente distribuido de la arquitectura J2EE permite a los componentes de un sistema correr en diferentes máquinas para desempeñar una tarea común. La naturaleza multicapa de la arquitectura J2EE soporta un ambiente distribuido para la aplicación. Por ejemplo, mientras los componentes que perteneces a diferentes capas pueden correr en diferentes máquinas, ellos también pueden correr en la misma máquina. Podemos notar que hay mucha flexibilidad aquí. Se puede correr componentes que pertenecen a todos las capas en una sola máquina. No obstante es

un escenario impractico, además de que no se estaría tomando ventaja del ambiente distribuido que proporciona J2EE.

Portabilidad

Portabilidad, en general se refiere a la habilidad de una aplicación de correr en cualquier plataforma. El lema del lenguaje java es **"write once, run anywhere**. Esto significa que una aplicación escrita en Java puede correr en cualquier Sistema Operativo. En otras palabras, Java ofrece independencia de Sistema Operativo. J2EE ofrece portabilidad doble: independencia de Sistema Operativo e independencia de vendedor. Independencia de Sistema Operativo es el resultado de que J2EE es basada en la plataforma Java, mientras que independencia de vendedor se refiere a la habilidad de las aplicaciones J2EE de ser desplegadas y ejecutadas en diferentes implementaciones de la plataforma J2EE por diferentes vendedores.

Interoperabilidad

Interoperabilidad se refiere a la habilidad de los componentes de software escritos en un lenguaje y ejecutadas en un ambiente como J2EE de comunicarse con los componentes de software escritos en otro lenguaje y corriendo en diferentes ambientes como .NET. La interoperabilidad de J2EE con plataformas diferentes a Java es gracias al soporte de los protocolos estándares de comunicación de Internet como son TCP/IP, HTTP, HTTPS y XML.

Escalabilidad

Esta propiedad se refiere a la capacidad de un sistema de poder soportar un incremento en el número de clientes sin degradar su confiabilidad y desempeño. También significa que si hay un gran incremento en el número de clientes del sistema, solo se necesitar hacer relativamente pocas adiciones al sistema para satisfacer la demandas de los clientes.

Alta disponibilidad y Alto desempeño

La arquitectura Multicapa de J2EE también soporta alta disponibilidad y alto desempeño. Estas características pueden ser implementadas a través de clusters y balaceo en la carga. Los servidores pueden ser añadidos dinámicamente para satisfacer las demandas. Es importante recalcar que J2EE es un sistema de software y que por tanto la alta disponibilidad y alto desempeño en este contexto son atributos del software.

Simplicidad

Gracias a que las tareas de implementación a nivel de infraestructura de servicios como soporte para transacciones y sockets son hechas por el servidor J2EE, el desarrollador solo tiene que codificar lógica de negocio, lo cual hace simple el desarrollo de aplicaciones J2EE.

1.2.2 Contenedores de la plataforma J2EE

Antes de que un componte pueda ser ejecutado, debe ser ensamblado dentro de una aplicación J2EE y desplegado dentro del contenedor. En otras palabras, un componente de una aplicación J2EE puede ser solo ejecutado por un contenedor y no sin el. Un contenedor J2EE mantiene solo un tipo de componentes de aplicación J2EE, proveyendo a los componentes el ciclo de ejecución que ellos requieren, desde su creación hasta su muerte. Por ejemplo, el contenedor web contiene los componentes de la capa web, y el contenedor de EJB contiene los componentes de la capa EJB.

Algunas de las funciones que los contenedores desempeñan son listadas a continuación:

- Proveer métodos para los componentes de la aplicación para que interactúen uno con otro, así los contenedores son los mediadores de la comunicación.
- El contenedor provee la API necesaria que ocupan los componentes
- El contenedor ofrece servicios para los componentes tales como manejo de transacciones, seguridad, conectividad remota, etc.

Implementando el contenedor toda la infraestructura necesaria tal como conectividad remota para todas las aplicaciones J2EE, hace el desarrollo de una aplicación mucho más simple y fácil. Los desarrolladores de aplicaciones J2EE se pueden ahora solo enfocarse en la presentación y lógica de negocio.

1.3 Tecnologías y servicios empleados para desarrollar y soportar aplicaciones J2EE

La plataforma J2EE ofrece soporte de principio a fin en aplicaciones, desde el cliente hasta la capa de información empresarial. Las aplicaciones J2EE son por lo general complejas y sofisticadas. No obstante, un servidor J2EE simplifica el desarrollo, despliegue y ejecución de una aplicación. JBoss, Websphere de IBM y Weblogic de BEA son ejemplos de servidores de aplicaciones, y todos ellos definen el mismo estándar definido por la especificación J2EE hecha por SUN. Una aplicación J2EE puede ser desplegada y ejecutada en un servidor J2EE de cualquier vendedor, este es un principio llamado **"write once, deploy anywhere"**, citado previamente .

1.3.1 Tecnologías J2EE

Para desarrollar aplicaciones empresariales, es necesario utilizar una variedad de tecnologías, las cuales funcionan en distintas capas. A continuación se mencionan estas tecnologías:

Tecnologías Cliente

Un cliente J2EE es un componente de software que se encuentra en la capa cliente. Hay dos tipos de clientes:

- ***Cliente de Aplicación***: Un cliente de aplicación se puede comunicar directamente con un componente de software en la capa EJB ó con un componente en la capa web usando HTTP. Un cliente de aplicación puede tener una interfaz de línea de comando o una interfaz gráfica de usuario (GUI). Clientes de aplicación son desarrollados usando el lenguaje Java.

- ***Cliente Web*** : Un cliente web es un componente de software en la capa cliente que usa HTTP para comunicarse con los componentes en la capa web. Puede ser un programa Java que abre una conexión HTTP, o un browser.

Tecnologías de Presentación (Tecnologías Web)

Estas tecnologías son usadas para desarrollar componentes que trabajan en la capa web o también conocida como capa de presentación. Estos componentes reciben la petición del cliente y se comunican con los componentes en la capa EJB, o ellos pueden directamente interactuar con la base de datos en la capa de EIS. Ellos se encargan de dar una respuesta, por ejemplo una página web, y la envían al cliente web. Estos componentes de software son desarrollados usando las tecnologías de Servlets y JSP. Los Servlets son clases Java que se encargan de dar respuesta a las peticiones hechas por un cliente web a través de protocolo HTTP. Una JSP es un documento el cual contiene scripts hechos en Java, que es traducido automáticamente a un Servlet por el contenedor de JSP y Servlets.

Tecnologías de lógica de Negocio

Estas tecnologías son usadas para desarrollar componentes de software para la capa de negocios o capa EJB. Los componentes construidos usando esta tecnología implementan la lógica de negocio de la aplicación. En J2EE, estos componentes de negocio son desarrollados usando la tecnología llamada Enterprise JavaBeans.

Tecnologías para interactuar con base de datos

Los componentes de software en la capa web o en la capa EJB usan la API para base de datos si ellos se quieren comunicar directamente con la base da datos. La plataforma J2EE requiere el uso de la API JDBC para comunicarse con la base de datos.

Es interesante destacar que para que los componentes de distintas capas puedan estar ejecutándose en distintas máquinas o se puedan comunicar con componentes de otra capa, es necesario ciertos servicios, los cuales se ven a continuación.

1.3.2 Servicios proporcionados por J2EE

La especificación J2EE requiere un conjunto de servicios standard que un vendedor de productos J2EE debe implementar. Esos servicios son usados para soportar las aplicaciones J2EE. Algunos de estos servicios que se presentan aquí son:

Servicio de nombres (Naming Service)

Debido a que la plataforma J2EE ofrece un ambiente distribuido, los componentes y servicios pueden residir en múltiples máquinas. Consecuentemente, se necesita un mecanismo de búsqueda que es requerido para localizar componentes y servicios. Java Naming and Directory Interface (JNDI) proporciona este mecanismo. Por ejemplo, un cliente usaría JNDI para encontrar un servicio en otra máquina.

Servicio para Transacciones

El manejo de transacciones para los componentes de la aplicación es una tarea importante. La API Java Transaction (JTA) nos ayuda a desempeñar esta tarea. Por ejemplo, supongamos que múltiples clientes intentan acceder a la base de datos concurrentemente. JTA nos ayudaría a mantener la integridad de los datos durante estos accesos concurrentes a la base de datos.

Servicio de mensajes

Aplicaciones empresariales necesitan intercambiar mensajes tales como notificaciones de ciertos eventos a otras aplicaciones. Java Message Service (JMS) facilita el envío de mensajes en ambos modelos: punto a punto (*point to point*) y publicación por suscripción (*publish-subscribe*). En el modelo point to point, un mensaje es enviado solo por un transmisor y es recibido solo por un receptor, mientras que en el modelo publish-subscribe, un mensaje enviado por un transmisor puede ser recibido por múltiples receptores llamados subscritores.

Servicio de correo

El correo es un servicio el cual es comúnmente necesario en aplicaciones empresariales, por lo cual la Java Mail API implementa toda la funcionalidad requerida.

Servicio de acceso remoto

En ambientes distribuidos, un componente en una máquina puede necesitar invocar un método de un componente en otra máquina, lo cual es conocido como invocación remota. El estándar RMI (Remote Method Invocation) oculta los detalles de red, proveyendo a los objetos que invocan métodos remotos transparencia en la invocación. La API de Java proporciona soporte para RMI.

Servicio de procesamiento de XML

El lenguaje de marcado extendido (XML) es una parte integral de aplicaciones J2EE. La API de Java para XML (JAXP) es usada para escribir e interpretar archivos XML. JAXP provee soporte para herramientas de XML tales como la API simple para XML (SAX), Modelo de Objeto de Documento (DOM), y XML Stylesheet Language para Transformations (XSLT).

Servicio de Seguridad

La seguridad en una aplicación empresarial es implementada comúnmente para controlar la autenticación y autorización de un usuario a la misma. Java Authentication aun Authorization Service (JAAS) provee este tipo de seguridad por ofrecer un mecanismo para autenticación de usuarios o un grupo de usuarios que quieren acceder a la aplicación.

1.4 Roles en J2EE

Cuando se desarrolla una aplicación empresarial J2EE, existen varios roles que desarrollan ciertas tareas como desarrollar los componentes, desplegarlos, ensamblar los componentes de la aplicación etc. Las responsabilidades de quien implementa cada cosa en una aplicación es llamado ***roles de implementación***. A continuación se describen los roles.

Proveedor de productos J2EE

El proveedor de productos J2EE se refiere al vendedor que implementa toda la plataforma J2EE con todos los contenedores y servicios requeridos por la especificación. Este proveedor tambien suministra las herramientas de despliegue y administración. Ejemplo de proveedores son BEA, IBM y Sun

Proveedor de Herramientas

Un proveedor de herramientas desarrolla y comercia herramientas para el desarrollo, empaquetamiento, despliegue, administración, y monitoreo para aplicaciones. El proveedor de productos J2EE usualmente también proporciona las herramientas.

Proveedor de componentes de aplicación

Este es el desarrollador (persona o compañía) que desarrolla los componentes que se ejecutaran el contenedor Web o en el contenedor de EJB.

- *Desarrollador de componentes Web*: La responsabilidad de un desarrollador de componentes web es implementar los componentes que crearan la presentación para un cliente. Los siguientes son los componentes que desarrolla: JSP, Servlets, HTML, y clases Java, incluyendo JavaBeans. El desarrollador componentes web también escribe descriptores de despliegue y crea los archivos Web (WAR).

- *Desarrollador de componentes EJB*: La responsabilidad de un desarrollador de componentes EJB (persona o compañía) es desarrollar Enterprise JavaBeans, escribir el descriptor de despliegue, Y crear los archivos Java (JAR), llamados archivos *ejb-jar*.

Ensamblador de la aplicación

La responsabilidad de un ensamblador de aplicación es ensamblar los componentes (componentes web y componentes EJB) dentro de una aplicación J2EE completa. Esto involucra empaquetar el archivo WAR y el archivo ejb-jar dentro de un archivo empresarial (EAR).

Persona encargada del despliegue de la aplicación

Después de que la aplicación ha sido ensamblada, necesita ser desplegada dentro de un ambiente de ejecución antes de que pueda se utilizada. El deployer desempeña esta tarea siguiendo los siguientes pasos.

1. Instalación: Mover el archivo ejb-jar al servidor J2EE.
2. Configuración: Configurar la aplicación para el ambiente de ejecución, adecuando los descriptores de archivo.
3. Ejecución: Iniciar la aplicación.

Administrador de la aplicación

Un administrador de la aplicación es el responsable del monitoreo de la aplicación, y de configurar los servidores de aplicaciones para su mejor desempeño.

1.5 Resumen

La plataforma J2EE es una especificación para el desarrollo de aplicaciones distribuidas basadas en componentes. La plataforma J2EE la podemos estudiar en base a su arquitectura, servicios y tecnologías proporcionadas, y roles de implementación.

La arquitectura J2EE es la forma en la que las aplicaciones J2EE están construidas. Encontramos que la arquitectura J2EE esta conformada de varias capas, donde cada capa desempeña un papel específico en la aplicación. Entre las capas que encontramos estan la capa cliente, capa web, capa de negocios o EJB, y capa de Sistemas de Información Empresarial.

Los servicios y tecnologías proporcionadas por la plataforma J2EE son necesarias para desarrollar los componentes que se van a ejecutar en cada capa y proporcionar una comunicación entre ellos.

Por otra parte, los roles definen quien hace que cosa durante el desarrollo, despliegue y administración de la aplicación empresarial.

Introducción a los Enterprise Java Beans

2.1 Introducción

En el capítulo pasado, describimos a grandes rasgos la arquitectura J2EE, la cual esta compuesta de múltiples capas, entre ellas la capa web y de negocios. La capa de negocios de la plataforma J2EE, es la que se encarga de implementar la lógica del negocio. Es en la capa de negocios donde trabajan nuestros componentes EJB, los cuales necesitan un ambiente de ejecución para poder trabajar, este ambiente es proporcionado por el contenedor de EJB. Entre las principales características de la arquitectura EJB se encuentra que esta basada en componentes, se ejecuta en el lado del servidor, es una arquitectura distribuida, son escalables y seguros, proporciona un ambiente transaccional, estandariza el desarrollo, despliegue y ejecución de aplicaciones empresariales.

Podemos destacar que la arquitectura EJB esta compuesta de dos partes principales: ***componentes EJB (enterprise beans) y el contenedor EJB***.

En este capítulo nos enfocaremos a describir la arquitectura EJB, los componentes EJB que conforman esta arquitectura, además de todos los roles involucrados en el desarrollo, despliegue y administración de los componentes EJB. Comenzaremos este capítulo dando un breve repaso a las arquitecturas distribuidas.

2.2 Arquitecturas de objetos distribuidas

Para comprender EJB, se necesita comprender como trabajan los objetos distribuidos. Todos los protocolos de objetos distribuidos son construidos con la misma arquitectura, que es diseñada para hacer que un objeto en una computadora luzca como si estuviera residiendo en una computadora diferente.

Las arquitecturas de objetos distribuidas están basadas en una capa de comunicación de red que es muy simple. Esencialmente, hay tres partes en estas arquitecturas: el objeto de negocio (bussines objet), el skeleton, y el stub.

El objeto de negocio reside en el servidor, y es una instancia de una clase que modela el estado y lógica de negocio de algo del mundo real, tal como una persona, una orden o cuenta. Cada clase que representa a un objeto de negocio tiene una clase **stub** y **skeleton** construidas específicamente para ese objeto de negocio.

El ***stub*** y el ***skeleton*** son los responsable de hacer que el objeto de negocio en el servidor luzca como si estuviera ejecutándose localmente en la máquina cliente. Esto se logra a través del protocolo de invocación remota de métodos (RMI). El protocolo RMI se usa para comunicar la invocación de método sobre la red. CORBA, Java RMI, y Microsoft .NET son algunos ejemplos que usan protocolos RMI. Cada instancia del objeto de negocio en el servidor, es encapsulado por una instancia de la clase **skeleton**. Entre las tareas del skeleton están la de permanecer escuchando a través de un puerto por peticiones provenientes del stub, que reside en la máquina cliente y esta conectado al skeleton a través de la red. El stub actúa como sustituto del objeto de negocio en el cliente y es el responsable de comunicar las peticiones del cliente al objeto de negocia a través del **skeleton**.

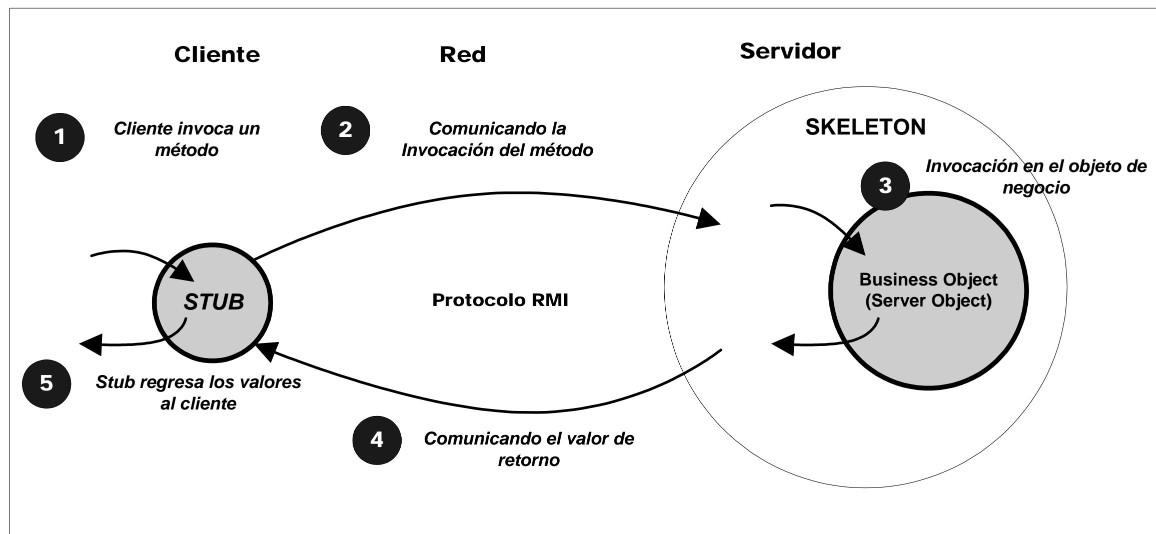


Figura 2.1 Elementos de una Arquitectura de objetos distribuida (Stub y Skeleton)

El objeto de negocio implementa una interfaz pública que declara todos los métodos de negocio. El stub implementa la misma interfaz que el objeto de negocio, pero los métodos del stub no contienen la lógica del negocio, en vez de eso implementan cualquier operación de red requerida para enviar las peticiones al objeto de negocio y recibir los resultados. Cuando un cliente invoca un método del objeto de negocio en el stub, la petición es comunicada a través de la red al skeleton. Esta petición contiene el método que se invocó, y sus parámetros. Después de que el skeleton descubre que método se invocó y sus parámetros, invoca al correspondiente método de negocio en el objeto de negocio. Cualquier valor que es regresado del método que se invoco en el objeto de negocio es envío de regreso al stub por el skeleton. El stub luego regresa el valor al cliente, de modo que pareciera que se proceso la invocación del método localmente.

2.3 Arquitectura EJB

EJB es una arquitectura basada en componentes que soporta el desarrollo de aplicaciones empresariales distribuidas y portables a distintos sistemas operativos, además manejo transaccional y automatización de la seguridad. La arquitectura EJB tiene tres piezas principales:

- **Componentes de software**: Los Enterprise JavaBeans, también llamados enterprise beans o simplemente beans.
- **Contenedor EJB**: contiene y maneja a los beans, además de ofrecer determinados servicios.
- **Servidor EJB**: aloja al contenedor EJB y ofrece servicios a nivel de sistema como lo es el manejo de subprocesos.

Actualmente, la especificación de EJB no hace una clara distinción entre el contenedor EJB y el servidor EJB.

La arquitectura EJB define tres tipos de enterprise beans:

- **Beans de Sesión (Session Beans)**: sirven para representar un proceso del negocio, por ejemplo un carrito de ventas.
- **Beans de Entidad (Entity Beans)**: representan un registro o entidad en una base de datos, por ejemplo, un entity bean podría representar un vendedor en la base de datos.
- **Beans para el Manejo de Mensajes (Message-Driven Beans)**: se encargan del manejo de mensajes del servicio de mensajería de Java (JMS).

2.3.1 El contenedor y sus requerimientos

Una de las principales funciones del contenedor es mediar entre las peticiones hechas por el cliente al bean. El contenedor también provee soporte en tiempo de ejecución para los beans y ofrece un conjunto de servicios que necesita el bean para su correcto funcionamiento. A continuación se listan algunos de los servicios ofrecidos por el contenedor.

Gestión de transacciones

Las aplicaciones por lo general requieren soporte realizar transacciones. Una transacción es un conjunto de operaciones, generalmente en una base de datos, que son tratadas como atómicas. La plataforma EJB soporta dos tipos de transacciones: **local y distribuida**. Una transacción local usualmente involucra a una base de datos, mientras que una transacción distribuida involucra múltiples bases de datos en varias máquinas. Para ofrecer el servicio de manejo de transacciones, el contenedor usa la API para transacciones de Java (JTA).

Gestión de la seguridad

El contenedor ofrece el servicio de control de acceso y autenticación. Estos servicios son configurados en el descriptor de archivo del enterprise bean.

Gestión del ciclo de vida del Enterprise Bean

El contenedor es el encargado de proveer servicios tales como creación de objetos, manejo de hilos, y la destrucción de objetos. El ciclo de vida de un bean es de acuerdo al tipo que pertenece.

2.3.2 Componentes EJB

EJB es una arquitectura base en componentes, donde **componente** se refiere a piezas reusables de software que son escritas de acuerdo a un standard. En EJB, los enterprise beans son escritos en Java, y son usados para implementar lógica de negocio de una aplicación empresarial. Es importante destacar que EJB y enterprise beans no es lo mismo. ***Enterprise JavaBeans (EJB) se refiere a la arquitectura de la capa de negocio en la especificación J2EE, y enterprise beans son los componentes en esta arquitectura.*** El término EJB es tratado como si su significado fuera el de enterprise beans.

Entre las características principales que encontramos en los Enterprise Beans están las siguientes:

Controlados por el Contenedor

Un enterprise bean es creado, manejado y destruido por el contenedor EJB. No puede ejecutarse sin el contenedor. En otras palabras, no tiene un método main(). Además, el contenedor actúa como un mediador entre el bean y el cliente, ya que el contenedor se encarga de la seguridad y el manejo transaccional.

Write once, deploy anywhere (WODA)

Debido a que los enterprise beans son escritos en Java, adquieren automáticamente la independencia de plataforma: "write once, run anywhere". No obstante, como los enterprise beans pueden ser ejecutados en cualquier contenedor EJB que cumpla con la especificación, estos pueden ejecutarse en cualquier servidor de aplicaciones que contiene un contenedor de EJB como Websphere, Weblogic y JBoss, de modo que se escriben una vez y se despliegan en cualquier servidor de aplicaciones: **"WODA"**.

Reusabilidad

Un enterprise bean, una vez escrito puede se usado en múltiples aplicaciones sin tener que hacer ningún cambio al código ni tener que recompilarlo, lo cual favorece el reuso de estos componentes. Estas capacidades vienen de que son independientes de plataforma (write once, run anywhere), independientes del servidor (write once, deploy anywhere), y de que se pueden configurar usando su descriptor de archivo.

2.3.3 JavaBeans y Enterprise JavaBeans

Es común confundir el término JavaBeans y Enterprise JavaBeans, los dos tienen significados diferentes, y lo único que comparten es que están escritos en Java. JavaBeans y Enterprise JavaBeans son dos diferentes tecnologías Java. Mientras que los JavaBeans son clases Java escritas de acuerdo a ciertas reglas y ejecutadas en una JVM, los componentes Enterprise JavaBeans necesitan un contenedor para su ejecución y despliegue. Las diferencias entre ambos se presentan a continuación:

- Los JavaBeans se ejecutan en una JVM, mientras que los componentes Enterprise JavaBeans necesitan un contenedor para su ejecución.

- Los JavaBeans y los componentes Enterprise JavaBeans son escritos en Java, no obstante, el comportamiento de los componentes Enterprise JavaBeans puede ser modificado sin cambiar una sola línea de código, utilizando los descriptores de despliegue.

- Los JavaBeans son más simples, mientras que los enterprise beans puede ser mucho más sofisticados e implementan a menudo la lógica de negocio que se ejecuta de lado del servidor.

2.4 Roles específicos para EJB

La arquitectura EJB nos provee los elementos básicos para construir aplicaciones distribuidas. Entre las ventajas de las aplicaciones distribuidas que se basan en una arquitectura de componentes, es que se pueden repartir responsabilidades entre las partes o individuos que conforman una organización. La especificación EJB define seis distintos roles en el ciclo de vida de una aplicación EJB. Estos roles, dependiendo del tamaño de la aplicación pueden ser ejecutados por un mismo individuo. Los seis roles definidos en la especificación EJB son los siguientes:

- Enterprise bean provider
- Applicaction assembler
- Bean deployer
- EJB server provider
- EJB container provider
- System administrator

Los roles son descritos a continuación:

Enterprise bean provider

El enterprise bean provider es el encargado de implementar la lógica de negocio de la aplicación. Desarrolla las clases java que necesita el enterprise bean, las cuales contienen los métodos de negocio. También implementa las interfaces java que el cliente usará para interactuar con el bean, y construye el descriptor de despliegue.

Finalmente, todas las clases e interfaces que construyó, las empaqueta junto con el descriptor de despliegue en un archivo JAR, llamado ejb-jar. Un archivo ejb-jar puede llamarse como sea, y puede contener más de un enterprise bean, pero sólo un descriptor de despliegue, el cual se debe llamar ejb-jar.xml.

Application assembler

El application assembler se encarga de ensamblar la aplicación empresarial, uniendo todos los componentes que puede recibir. Por ejemplo, puede ensamblar los enterprise beans para que trabajen en conjunción con las JSP y Servlets. También modifica algunos datos de configuración en el descriptor de despliegue ejb-jar.xml, y algunas veces en el web.xml de una aplicación web. Todo esto lo hace antes de que se despliegue la aplicación empresarial.

Bean deployer

El bean deployer obtiene los enterprise bean en la forma de un archivo ejb-jar del application assembler o del bean provider y los despliegua en el contenedor EJB. El bean deployer es por lo general un experto en el manejo del servidor de aplicaciones que contiene al contenedor EJB.

EJB server provider

El servidor EJB proveé el ambiente en el cual el contenedor se ejecuta. Por ejemplo, se encarga del manejo de procesamiento múltiple y manejo de carga.

EJB container provider

La responsabilidad de este rol es implementar el soporte necesario que los enterprise beans requieren en tiempo de ejecución, además de proveer servicios tales como gestión de seguridad, gestión de transacción y mantenimiento del pool de los beans.

2.5 Desarrollo de un enterprise bean

Para poder construir un enterprise bean, es necesario hacer 5 cosas:

1. Escribir la clase del bean, el cual contendrá todos los métodos del negocio.

2. Escribir dos interfaces para el bean: ***home y component***.

    a. Una interface home, que funciona como factoría de objetos que implementan la interface EJBObject.
    b. La interface component hereda de la interface EJBObject, y a través de los objetos que implementen la interface component es posible comunicarse con el enterprise bean.

3. Crear un archivo XML de configuración llamado **deployment descriptor**, éste archivo le indica al servidor de que tipo es el bean y como debe ser manejado. El nombre de este archivo debe ser **ejb-jar.xml**.

4. Empaquetar el bean, las interfaces y el descriptor de despliegue en un archivo JAR mejor conocido como **ejb-jar**, el cual podemos nombrar como sea.

5. Desplegar el enterprise bean en el servidor. La forma de desplegar el enterprise bean depende de cada vendedor.

En esta sección desarrollaremos un enterprise bean, el cual se encarga de darnos una cita aleatoria de Albert Einsten. En posteriores capítulos se ahondará acerca de cada parte del desarrollo del enterprise bean.

> **Nota:** *En el capítulo de despliegue de enterprise beans usando JBoss se muestra como desplegar y ejecutar todos los ejemplos de enterprise beans que se van desarrollando durante los capítulos.*

Paso 1 "Escribir el bean"

La clase del bean contiene los métodos del negocio, que un cliente puede llamar, además de implementar unos métodos adicionales que controlan su ciclo de vida. De acuerdo a tipo de bean que queramos desarrollar es la interfaz que debemos implementar.

Bean : QuoteBean.java

```
package com.quote;

import javax.ejb.*;

public class QuoteBean implements SessionBean{
    private String []quotes = {
        "Before God we are all equally wise - and equally foolish",
        "I never think of the future - it comes soon enough",
        "Imagination is more important than knowledge...",
        "Reality is merely an illusion, albeit a very persistent one",
        "The important thing is not to stop questioning",
        "The secret to creativity is knowing how to hide your sources",
        "Science without religion is lame, religion without science is blind",
        "Everything that is really great and inspiring is created by the individual who can labor in freedon"
    };
    public void ejbCreate(){
        System.out.println("ejecutando ejbCreate()");
    }
    public void ejbActivate(){
        System.out.println("ejecutando ejbActivate()");
    }
    public void ejbPassivate(){
        System.out.println("ejecutando ejbPassivate()");
    }
    public void ejbRemove(){
        System.out.println("ejecutando ejbRemove()");
    }
    public void setSessionContext( SessionContext ctx ){
        System.out.println("ejecutando setSessionContext()");
    }

    // Método de negocio
    public String getQuoteEistein(){
        System.out.println("ejecutando getQuoteEistein");
        int random = (int)(Math.random() *  quotes.length);
        return quotes[random];
    }
}
```

Paso 2 "Escribir las interfaces : home y component"

El cliente del bean usa las interfaces home y component para obtener acceso a los métodos de negocio del bean. El cliente usa la home interface que hereda de la EJBHome para obtener una referencia a la interfaz del componente, el cual extiende de la interface EJBObject. La component interface es usada para invocar los métodos del negocio en el bean, porque el bean expone sus métodos de negocio en la interfaz de componente. Un cliente nunca accede directamente al bean, siempre invoca los métodos de negocio a través de la component interface.

A través de la home interface obtenemos una referencia a la interfaz component interface, a través de la cual invocaremos los métodos que deseamos que ejecute el bean. Podemos notar que la principal tarea de la home interface es manejar las referencias de los componentes de interface.

Home interface: QuoteHome.java

```
package com.quote;

import javax.ejb.*;
import java.rmi.RemoteException;

public interface QuoteHome extends EJBHome
{

   public QuoteComponent create() throws CreateException, RemoteException;

}
```

En la component interface, se declara el método **getQuoteEisten()**, que está en la clase QuoteBean, esto se debe a que el cliente de este enterprise bean solo puede invocar a los métodos que se encuentran en la component interface.

Component interface : QuoteComponent.java

```
package com.quote;

import javax.ejb.*;
import java.rmi.RemoteException;

public interface QuoteComponent extends EJBObject
{

   public String getQuoteEistein() throws RemoteException;

}
```

Paso 3 "Crear el descriptor de despliegue"

En el descriptor de despliegue se encuentra la información de cómo se debe desplegar el entrerprise bean, datos de configuración que necesita el bean e información acerca de seguridad. Este descriptor siempre debe llamarse **ejb-jar.xml**.

Deployment Descriptor : ejb-jar.xml

```
<?xml version="1.0"?>
<ejb-jar
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                  http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
    version="2.1">
 <enterprise-beans>
   <session>
      <display-name>QuoteBean</display-name>
      <ejb-name>QuoteBeanSession</ejb-name>
      <home>com.quote.QuoteHome</home>
      <remote>com.quote.QuoteComponent</remote>
      <ejb-class>com.quote.QuoteBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
   </session>
 </enterprise-beans>
</ejb-jar>
```

Entre las principales etiquetas del descriptor de despliegue se encuentra <ejb-name> la cual nos sirve para darle un nombre al bean, y este nombre es utilizado por lo general por el servidor EJB para identificar al enterprise bean.

Las etiquetas <home> y <remote> me sirven para indicar cuales son las interfaces home y component de este enterprise bean.

Otra etiqueta importante es <session>, la cual indica que el enterprise bean es un Bean de Sesión. La etiqueta <session-type> indica que el enterprise bean es un bean de tipo stateless session bean, que se estudiara en el capitulo dedicado a beans de sesión.

Paso 4 "Crear el archivo *ejb-jar*"

En este paso, empaquetamos el bean, las interfaces y el descriptor de archivo dentro del archivo **ejb-jar** que se puede llamar de cualquier forma. Este archivo es el único que se necesita para distribuir nuestra aplicación, ya que las interfaces home y component son implementadas por el servidor de aplicaciones.

Dentro del archivo ejb-jar debemos tener un directorio **META-INF** en el cual colocamos nuestro descriptor de archivo ejb-jar.xml.

El archivo ejb-jar debe tener la siguiente estructura:

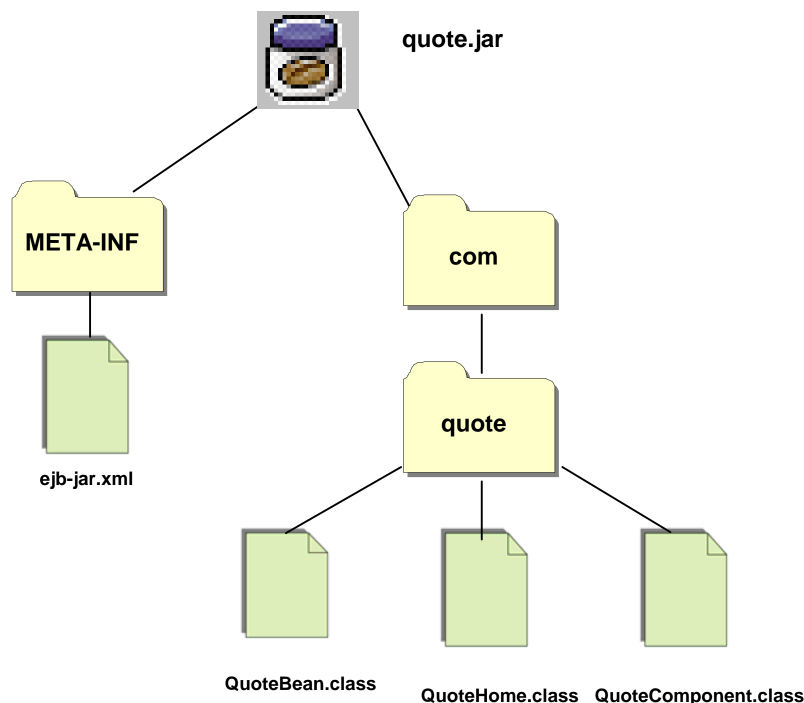


Figura 2.2 Estructura del archivo ejb-jar

Paso 5 "Desplegar el Enterprise Bean"

Una vez que se obtuvo el archivo *ejb-jar* es necesario desplegarlo en el servidor EJB para su ejecución. Esto depende específicamente del vendedor o proveedor del servidor.

Finalmente, necesitamos un cliente que solicite los recursos del enterprise bean. Este cliente tiene como característica que utiliza la interfaz de nombres y directorios de java (JNDI) para poder encontrar el recurso, en este caso el bean.

Cliente : QuoteClient.java

```
package com.clients;

import com.quote.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class QuoteClient {
   public static void main(String arg[]){
      try{
         Context ctx = new InitialContext();
         Object objref = ctx.lookup("quoter");
         QuoteHome home = (QuoteHome)PortableRemoteObject.narrow( objref, QuoteHome.class);
         QuoteComponent quote = home.create();
         System.out.println("La cita es " + quote.getQuoteEistein());
      }
      catch(Exception e){
         e.printStackTrace();
      }
   }
}
```

En esta clase utilizamos la clase InitialContext para obtener una entrada al contexto de JNDI, después efectuamos un búsqueda del recurso, en este caso llamado **"quoter"**, este nombre se estableció al desplegar el enterprise bean en el servidor EJB. El siguiente paso es utilizar el método ***narrow()*** de la clase PortableRemote el cual me regresa un objeto de la clase QuoteHome, debido a que el objeto objref trae un formato diferente al de la clase QuoteHome. Este método narrow() realiza las transformaciones necesarias para formar el objeto QuoteHome.

La posible salida de ejecutar la clase cliente es la siguiente:

Salidad de la clase QuoteClient

La cita es I never think of the future - it comes soon enough

2.6 Resumen

La arquitectura EJB es una arquitectura basada en componentes, los cuales son llamados ***componentes EJB*** ó ***enterprise beans***, estos se ejecutan dentro de un contenedor EJB el cual es contenido a su vez por un servidor.

Dependiendo de las necesidades del negocio, podemos encontrar tres tipos diferentes de enterprise beans: beans de sesión, beans de entidad, y beans para el manejo de mensajes.

Los beans de sesión se encargar de representar un proceso del negocio tal como un carrito de compras, registrar un venta, etc. Mientras que los beans de entidad representan un registro o entidad en una base de datos, tal como un consumidor. Los beans para el manejo de mensajes, son encargados del manejo del servicio de mensajería de Java.

Beans de Sesión (Session Beans)

3.1 Introducción

Como hemos visto, la arquitectura EJB ofrece tres tipos de beans para satisfacer las necesidades de una aplicación empresarial. Los beans de sesión son unos de los tres tipos de beans que puede desplegar un contenedor EJB. Un bean de sesión representa un proceso, tal como aplicación de un descuento a una compra o la verificación de una tarjeta de crédito.

Existen varios sabores para los beans de sesión: ***stateful y stateless***. Un *stateful session bean* sirve para mantener un estado conversacional con el cliente, ya que puede guardar determinados datos que un cliente le paso en una de las llamadas a un método. Un stateless session bean, al contrario de un stateful session bean, no puede mantener un estado conversacional con el cliente, de modo que no puede recordar información específica de un cliente.

Un bean de sesión es utilizado por un sólo cliente en un momento dado, después de que el cliente termina de usarlo puede ser destruido o ir al pool de beans, esto es dependiendo del sabor del bean de sesión.

En este capítulo mostraremos el funcionamiento de los beans de sesión, el ciclo de vida de un stateful y stateless session bean, y desarrollaremos un beans de cada sabor.

3.2 Beans de Sesión

Los beans de sesión nos sirven para efectuar un proceso de negocio de nuestra aplicación empresarial. El desarrollo de un bean de sesión implica la construcción de dos interfaces: Home y EJBObject, y una clase que implementa la inteface SessionBean. Cada uno de estos elementos juega un papel específico en la arquitectura de los beans de sesión.

3.2.1 Home Interface

El principal uso para lo que un cliente utiliza la home interface es obtener una referencia a un objeto que implementa la interface EJBObject o lo que es lo mismo, a un component interface. A través del component interface un cliente puede invocar métodos de negocio en bean, el cual nunca es accesado directamente por el cliente, sino solo a través del EJBObject.

A continuación mostramos los métodos que define la interface EJBHome.

<<interface>> **javax.ejb.EJBHome**
public EJBMetaData getEJBMetaData() public HomeHandle getHomeHandle() public void remove(Handle h) public void remove(Object key)

Es importante hacer notar que el contenedor es encargado de implementar todos los métodos en la home interface, y que es el contenedor el que se encarga de manejar la comunicación a través de la red, haciendo uso de los stubs, los cuales son necesarios, como vimos en el capítulo 2 para implementar el protocolo RMI.

Aquí esta una breve descripción de los métodos de la interface home:

getEJBMetaData() : Este método es utilizado por el cliente para obtener información acerca de la clase del bean.

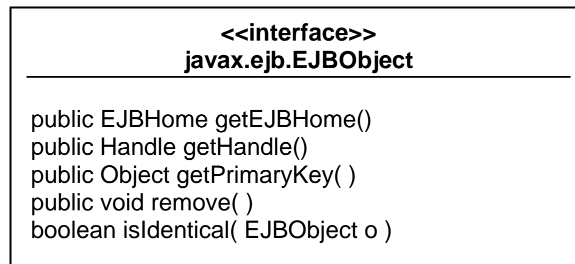
getHomeHandle() : Este método puede ser usado por un cliente para obtener un manejador de la interface home, de modo que si el cliente necesita obtener una referencia al objeto EJBHome de nuevo, no tenga que hacerlo a través de JNDI, si a través de este manejador.

remove(Handle h) : Un cliente de un bean de sesión puede llamar este método para informar al bean que el cliente no lo necesita más. Esto no necesariamente significa que la instancia del bean será removida: si el bean es un stateful session bean, la instancia será removida; si el bean es un stateless session bean, esta llamada no tiene efecto, ya que la instancia regresa al pool que es manejado por el contenedor.

remove(Object key) : Este método solo puede ser utilizado por beans de tipo Entidad, y recibe como argumento la llave primaria de la entidad, la cual será removida de la base de datos. Este método no tiene ningún significado para los bean de sesión, y si por algún motivo se intentara llamar en un bean de sesión, se lanzaría una excepción de tipo *javax.ejb.RemoveException*.

3.2.2 Component Interface

La interface Component que hereda de la interface EJBObject es utilizada para exponer los métodos de negocio que se escribirán en la clase del bean. El cliente nunca accede directamente al bean, solo lo hace a través de un objeto que implementa la interface EJBObject. La interface EJBObject es presentada a continuación:



La descripción de para que se utilizan estos métodos es la siguiente:

getEJBHome() : Este método regresa una referencia a un objeto de tipo EJBHome. Esto es útil cuando solo se cuenta con la component interface y se quiere obtener una referencia a objeto que implementa la interface EJBHome.

getHandle() : Este método regresa un manejador para el objeto de tipo EJBObject. Este manejador me permite en un momento dado recuperar los objetos EJBObject y EJBHome.

getPrimaryKey() : Este método regresa la llave primaria asociada con el objeto de tipo EJBObject. Debemos hacer notar que un bean de sesión no tiene primary key asociada, de modo que este método solo puede ser invocado por un bean de entidad.

remove() : Este método se puede comportar de una manera u otra dependiendo del sabor del bean de sesión. Si el bean es un stateful session bean, el contenedor removerá el bean, y con ello todos los datos asociados a ese cliente. Si el bean es un stateless session bean, el contenedor regresa el bean al pool, o lo puede remover para recuperar recursos tales como memoria.

isIdentical() : Este método sirve para probar si dos objetos de tipo EJBObject son idénticos, esto es, que ellos referencien a la misma instancia del bean. Si los objetos de tipo EJBObject son idénticos regresara true, de lo contrario false. Para los Stateless Session Bean, este método siempre regresara true si las referencias que el cliente esta comparando fueron obtenidas por la misma Home. En contraste, en el caso de un Stateful Session Bean, isIdentical() siempre regresara false, debido a que los beans de tipo Stateful tienen un determinado estado de acuerdo a los datos que han sido intercambiados con el cliente.

3.2.3 SessionBean interface

Para construir un bean de sesión debemos implementar la interface SessionBean. Esta interface contiene los métodos que el contenedor invoca para llevar a cabo el ciclo de vida del bean. Es en esta clase donde debemos declarar y definir los métodos de negocio del bean que el cliente podrá invocar a través del objeto que implementa la interface EJBObject. Además de estos métodos debemos declarar al menos un método llamado ejbCreate(), el cual también formara parte del ciclo de vida del contenedor.

La interface SessionBean se presenta a continuación:

<<interface>> **javax.ejb.SessionBean**
public void ejbActivate() public void ejbPassivate() public void ejbRemove( ) public void setSessionContext(SessionContext ctx)

Los cuatros métodos de esta interface controlan el ciclo de vida del bean de sesión, y tienen determinado comportamiento de acuerdo al sabor del bean de sesión.

La descripción de los métodos es la siguiente:

ejbActivate() : este método es invocado en un stateful session bean cuando el bean necesita ser reactivado después de que el contenedor lo guardo en un dispositivo de almacenamiento para liberar recursos.

ejbPassivate() : este método es invocada por el contenedor cuando requiere liberar recursos, este método debe tener tareas de limpieza de recursos, las cuales deben ser programadas por el desarrollador del bean.

ejbRemove() : este método se utiliza para las tareas de limpieza que se requieren antes de que el bean sea removido.

setSessionContext(SessionContext ctx) este método es invocado por el contenedor para proporcionarle información del cliente al bean a través de un objeto de tipo SessionContext.

3.3 Stateful Session Beans

Los stateful session beans sirven para representar un proceso en nuestro aplicación, además son capaces de mantener un estado conversacional con el cliente, esto quiere decir que podemos implementar un carrito de compras por ejemplo.

3.3.1 Desarrollo de un Stateful Session Bean

El desarrollo de un stateful session bean, como hemos visto implica la creación de dos interfaces, una que herede la interface EJBHome, y otra que hereda la interface EJBObject, y finalmente, una clase que implementa la interface SessionBean.

A continuación desarrollaremos un stateful session bean que simulará a grandes rasgos el proceso de compra de cerveza. Como primer paso escribimos la interface Home.

Home Interface : BeerHome.java

```
package com.beer;

import javax.ejb.*;
import java.rmi.*;

public interface BeerHome extends EJBHome{
   public BeerComponent create() throws CreateException, RemoteException;
}
```

Las reglas para escribir la interface home son las siguientes:

- Se debe importar javax.ejb.*, debido a que se usará la interface EJBHome y la excepción CreateException.
- Se debe importar java.rmi.RemoteException.
- Declarar un método create() con el component interface como valor de retorno. Los stateful session beans puede tener cualquier cantidad de métodos create sobrecargados, mientras que los stateless session bean solo pueden tener un método create sin argumentos.
- El método o los métodos create() deben declarar las excepciones CreateException y RemoteException.

A continuación mostramos la component interface y sus reglas:

Component Interface : BeerComponent.java

```
package com.beer;

import javax.ejb.*;
import java.rmi.*;

public interface BeerComponent extends EJBObject {
   public void setMarca( String marca ) throws RemoteException;
   public void setCantidad( int cantidad ) throws RemoteException;
   public double getCosto() throws RemoteException;
}
```

Las reglas para escribir la interface component son las siguientes:

- Se debe importar javax.ejb.*.
- Se debe importar java.rmi.RemoteException.
- Se debe heredar de javax.ejb.EJBObject.
- Los métodos de negocio declarados en la interface deben tener igual signatura o firma que los métodos de negocio que están definidos en la clase del bean, en nuestro ejemplo los métodos de negocio son *setMarca(String marca)*, *setCantidad(int cantidad)* y *getCosto()*.
- Todos los métodos se deben declarar public y no debe haber ningún método declarado static o final.

Las reglas presentadas anteriormente aplican igualmente a los stateless y stateful session beans.

La clase que define al bean de sessión es la siguiente:

Stateful Session Bean: BeerBean.java

```
package com.beer;

import javax.ejb.*;

public class BeerBean implements SessionBean {
   private String marca;
   private int cantidad;

   private SessionContext sessionContext;
   public void setSessionContext( SessionContext sctx ){
      sessionContext = sctx;
   }
   public void ejbPassivate(){
      System.out.println("Invocando ejbPassivate()");
   }
   public void ejbActivate() {
      System.out.println("Invocando ejbActivate()");
   }
   public void ejbRemove(){
      System.out.println("Invocando ejbRemove()");
   }
   public void ejbCreate(){
      System.out.println("Creando al objeto BeerBean");
   }
   // Métodos de negocio
   public void setMarca( String marca ){
      this.marca = marca;
   }
   public void setCantidad( int cantidad ){
      this.cantidad = cantidad;
   }
   // Imaginemos que el costo lo calculamos en base a información de una BD.
   public double getCosto(){
      return cantidad * 10.5;
   }
}
```

Las reglas para escribir la clase del bean son las siguientes:

- La clase debe implementar la interface javax.ejb.SessionBean.
- Por cada método create() en la home interface debe existir un método ejbCreate() en clase del bean.
- Debe tener un constructor sin argumentos.
- Los métodos de negocio se deben definir en esta clase, pero no tienen que empezar con "ejb", ya que todos los métodos de esta clase que empiezan con ejb controlan el ciclo de vida del bean.

Estas reglas son aplicables para los dos sabores de un session bean. El descriptor de despliegue de este bean se muestra enseguida:

Deployment Descriptor : ejb-jar.xml

```
<?xml version="1.0"?>
<ejb-jar
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                    http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
    version="2.1">
 <enterprise-beans>
   <session>
      <display-name>BeerBean</display-name>
      <ejb-name>BeerBean</ejb-name>
      <home>com.beer.BeerHome</home>
      <remote>com.beer.BeerComponent</remote>
      <ejb-class>com.beer.BeerBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Bean</transaction-type>
   </session>
 </enterprise-beans>
</ejb-jar>
```

En este descriptor de despliegue se encuentran definidas la siguientes etiquetas:

<home> : se utiliza para indicar cual es la home interface.

<remote> : se utiliza para indicar cual es la component interface

<ejb-class> : se utiliza para indicar cual es la clase del bean.

<session-type> : indica el sabor del bean de sesión, el cual puede ser *stateful* o *stateless.*

<transaction-type> : indica quien maneja las transacciones, si el contenedor o el bean.

Pasaremos a construir un cliente para este bean, el cual es definido como sigue:

Client: BeerClient.java

```
package com.client;

import com.beer.*;
import java.rmi.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import javax.swing.*;

public class BeerClient {
   public static void main( String arg[] ){
      try{
         Context ctx = new InitialContext();
         Object obj = ctx.lookup("beer");
         BeerHome beerHome = (BeerHome) PortableRemoteObject.narrow( obj, BeerHome.class );
         BeerComponent beerComponent = beerHome.create();
         String marca =
           JOptionPane.showInputDialog( null, "Cerveceria\nDame la marca de cerveza que quieres");
         beerComponent.setMarca( marca );
         int num = Integer.parseInt( JOptionPane.showInputDialog( null, "#Cantidad de cervezas"));
         beerComponent.setCantidad( num );
         JOptionPane.showMessageDialog( null, "El costo es " + beerComponent.getCosto());
         beerComponent.remove();
      }catch( Exception e ){
         e.printStackTrace();
      }
   }
}
```

Los pasos que el cliente siguió para poder invocar los métodos de negocio del bean son listados a continuación:

- El cliente hace un búsqueda del objeto home a través de JNDI, el cual es asociado al nombre **"beer"**.
- El servicio de JNDI regresa un stub al cliente del Objeto Home.
- El cliente llama al método create() a través del stub.
- El stub envía la llamada al objeto Home, que se encuentra en el servidor.
- El contenedor entonces crea un instancia del bean y del EJBObject.
- El stub para el EJBObject es regresado al cliente.
- El cliente utiliza el EJBObject para invocar los métodos de negocio del bean.

3.3.2 Arquitectura de un Stateful Session Bean

La arquitectura de un stateful session bean nos indica que esta formada por tres elementos: home interface, component interface y el bean. Esta arquitectura muestra el papel que desempeña cada uno de ellos, y su relación. A continuación se muestra esta arquitectura:

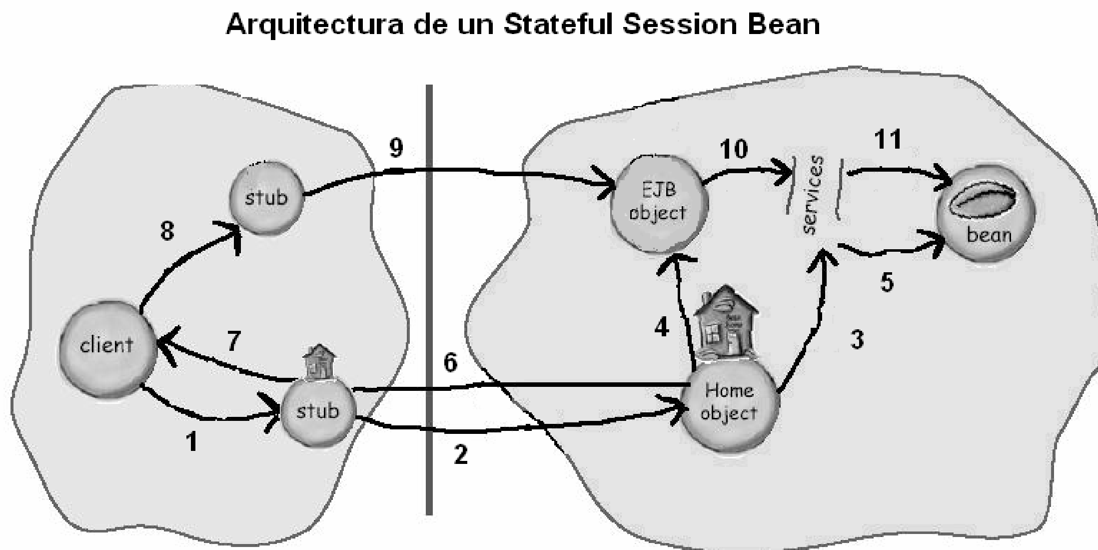


Figura 3.1 Arquitectura de un Stateful Session Bean

Pasos para invocar un método de negocio en el enterprise bean

Los pasos que se efectúan una vez obtenido un stub del Home Object a través de servicio JNDI son los siguientes:

1. El cliente llama al método create() en el Home stub.
2. El stub le dice al Home Object que el cliente quiere "crear" un bean.
3. El Home Object interactúa con los servicios del contenedor para crear un EJBObject y un bean.
4. El EJBObject es creado para el bean.
5. El bean es creado.
6. El Home Object regresa un stub del EJBObject al stub del Home Object.
7. El stub del EJBObject es regresado al cliente.
8. El cliente invoca un método de negocio en el stub del EJBObject.
9. El stub del EJBObject se comunica con el EJBObject notificándole la llamada del cliente.
10. Los servicios del contenedor en cuanto a seguridad, transacciones son ejecutados.
11. Se invoca el método de negocio en el bean.

3.3.3 Ciclo de vida de un Stateful Session Bean

El ciclo de vida de un stateful session bean consta de tres estados : no existe, listo y pasivo (passivated). La siguiente figura muestra su ciclo de vida:

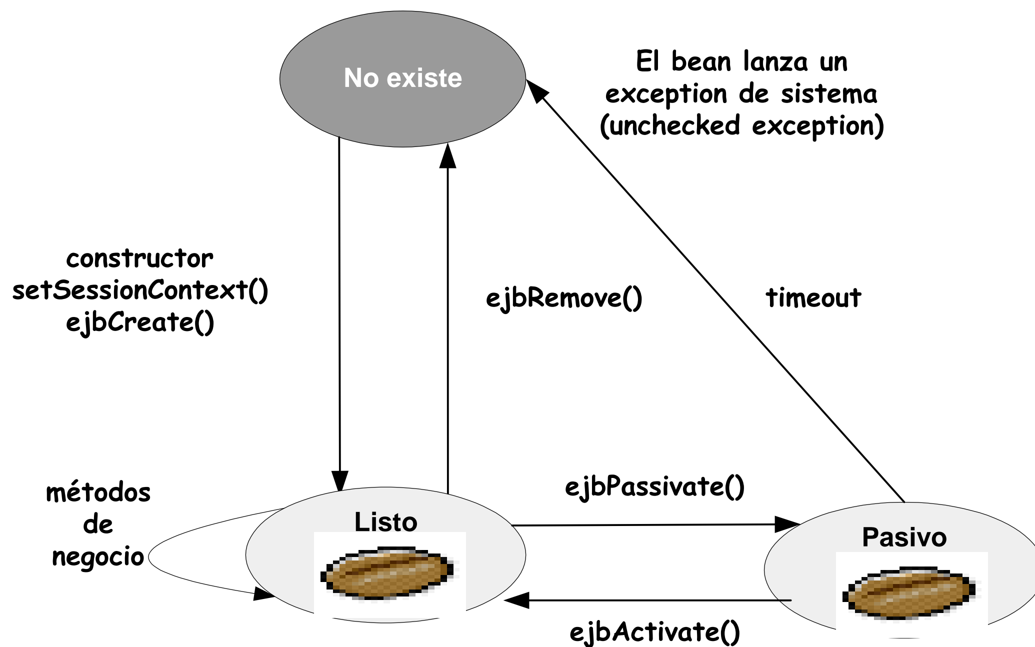


Figura 3.2 Ciclo de vida de un Stateful Session Bean

No existe El bean se encuentra en este estado cuando no ha sido creado o cuando el método ejbRemove() es invocado por el cliente o el tiempo de la sesión se ha agotado.

Listo El bean se encuentra en este estado cuando se ha invocado a su constructor, y los métodos setSessionContext() y ejbCreate() o cuando pasa del estado pasivado (Passivated) al estado listo. En este estado pueden ser invocados sus métodos de negocio por el cliente.

Pasivo Cuando el contenedor necesita liberar recursos tales como memoria, selecciona un stateful session bean para guardarlo con todo y sus atributos en el disco duro, siempre y cuando el cliente no este invocando los métodos de negocio en ese momento. Pasa del estado pasivo a listo, cuando el cliente vuelve invocar un método de negocio, así que el contenedor recupera al bean del disco duro e invoca al método ejbActivate() para activarlo.

3.4 Stateless Session Beans

Al igual que un stateful session bean, un stateless session bean sirven para representar un proceso de mi aplicación, pero un stateless session bean no puede mantener un estado conversacional con el cliente. Por lo general los stateless session bean se utilizan para realizar cálculos, como calcular el descuento a una compra de un determinado cliente.

Debido a que un stateless session bean no puede mantener un estado conversacional con el cliente, el contenedor puede usar diferentes instancias del bean para servir diferentes llamadas a métodos del mismo cliente, o puede usar la misma instancia del bean para ejecutar varias llamadas a métodos de diferentes clientes. Es importante destacar que un stateless session bean solo atiende a un cliente en un determinado momento, de eso se encarga el contenedor.

3.4.1 Desarrollo de un Stateless Session Bean

La construcción de un stateless session bean es básicamente de la misma forma que un stateful session bean.

En esta sección desarrollaremos un bean que nos muestra el descuento que debe aplicarse a un determinado cliente. En principio escribiremos nuestra interface Home, la cual se muestra a continuación:

Home Interface : DiscountHome.java

```
package com.discount;

import javax.ejb.*;
import java.rmi.*;

public interface DiscountHome extends EJBHome{
    public DiscountComponent create() throws CreateException, RemoteException;
}
```

Escribimos después la interface component:

Component Interface : DiscountComponent.java

```
package com.discount;

import javax.ejb.*;
import java.rmi.*;

public interface DiscountComponent extends EJBObject {
    public String[] getPersonasDescuento() throws RemoteException;
    public int getDescuento( String tipoPersona ) throws RemoteException;
}
```

Debemos hacer notar que tanto la interface home y component son implementadas por el contenedor, el cual se encarga de poner el código necesario para establecer un comunicación que utiliza principalmente stubs.

Definimos nuestro bean a continuación:

Stateless Session Bean : DiscountBean.java

```
package com.discount;

import javax.ejb.*;
import java.util.*;

public class DiscountBean implements SessionBean {
    // Imaginemos que los descuentos deacuedo al status de la persona se
    // obtiene de un base de datos.
    private Map discount = new HashMap();
    private SessionContext sessionContext;
    private String[] personasDescuento = {"adulto", "estudiante", "tercera edad", "niño"};

    public void ejbCreate(){
        discount.put( personasDescuento[0], new Integer(15));
        discount.put(personasDescuento[1], new Integer(50));
        discount.put(personasDescuento[2], new Integer(50));
        discount.put(personasDescuento[3], new Integer(25));
        System.out.println("Creando un Stateless Session Bean");
    }
    public void setSessionContext( SessionContext ctx ){
        this.sessionContext = ctx;
    }
    public void ejbActivate(){
    }
    public void ejbPassivate(){
    }
    public void ejbRemove(){
        System.out.println("Invocando ejbRemove");
    }
    // Métodos de negocio
    public String[] getPersonasDescuento(){
        return personasDescuento;
    }
    public int getDescuento( String tipoPersona ){
        return ((Integer)discount.get(tipoPersona)).intValue();
    }
}
```

Creamos un descriptor de despliegue, el cual nos indicará de que tipo es nuestro bean de sesión.

Deployment Descriptor : ejb-jar.xml

```
<?xml version="1.0"?>
<ejb-jar
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                    http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
    version="2.1">
 <enterprise-beans>
   <session>
      <display-name>Discount Bean</display-name>
      <ejb-name>Discount Bean</ejb-name>
      <home>com.discount.DiscountHome</home>
      <remote>com.discount.DiscountComponent</remote>
      <ejb-class>com.discount.DiscountBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
   </session>
 </enterprise-beans>
</ejb-jar>
```

Finalmente creamos una clase cliente la cual acceda a nuestro bean:

Client : DiscountClient.java

```
package com.client;

import javax.naming.*;
import javax.rmi.*;
import java.rmi.*;
import com.discount.*;

public class DiscountClient {
   public static void main( String arg[] ){
      try{
         Context ctx = new InitialContext();
         Object obj = ctx.lookup( "discount" );
         DiscountHome dh = (DiscountHome)PortableRemoteObject.narrow( obj, DiscountHome.class );
         DiscountComponent dc = dh.create();

         String personasDes[] = dc.getPersonasDescuento();
         System.out.println("Descuentos de acuerdo al tipo de personas");

         for( int i = 0; i < personasDes.length; i++ )
            System.out.println( personasDes[i] + " = " +
                        dc.getDescuento( personasDes[i]));
         dc.remove();

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

3.4.2 Arquitectura de un Stateless Session Bean

La arquitectura de un stateless session bean es similar a la de un stateful session bean. Esta arquitectura también cuenta con los tres elementos: home interface, component interface y el bean. En esta arquitectura, el contenedor por lo genera crea un pool de beans, donde almacena a los beans, y cada vez que el cliente invoca un método del bean, el contenedor saca del pool un bean, se ejecuta el método y vuelve a regresar al pool.

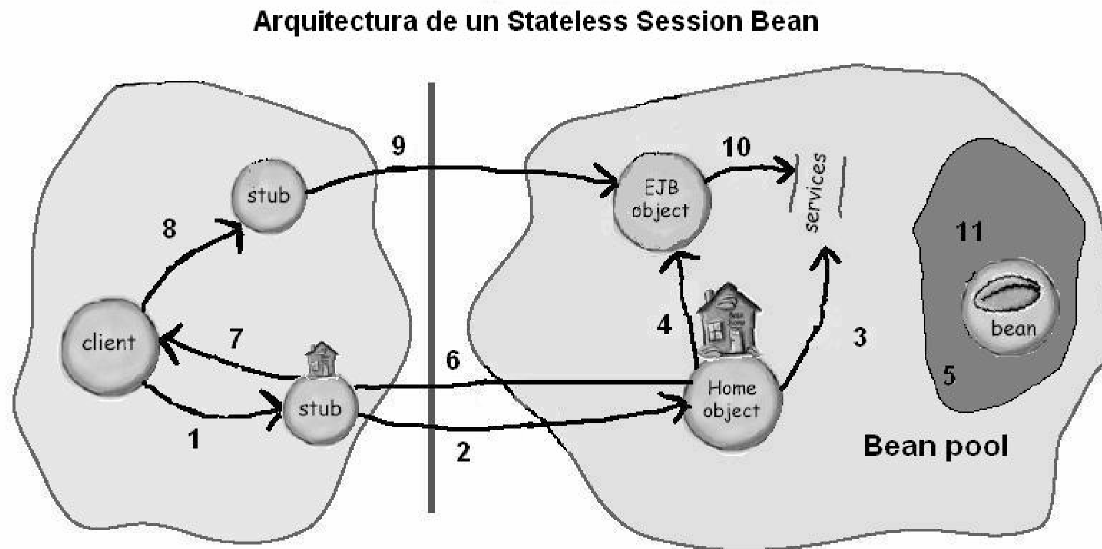


Figura 3.3 Arquitectura de un Stateless Session Bean

Los pasos que se efectúan una vez obtenido un stub del Home Object a través de servicio JNDI son los siguientes:

1. El cliente llama al método create() en el Home stub.
2. El stub le dice al Home Object que el cliente quiere “crear” un bean.
3. El Home Object interactúa con los servicios del contenedor para crear un EJBObject.
4. El EJBObject es creado, pero no esta asociado a un determinado bean.
5. El bean no es creado en este punto, ya que el contenedor pudo haberlo creado mucho antes de la llamada a create.
6. El Home Object regresa un stub del EJBObject al stub del Home Object.
7. El stub del EJBObject es regresado al cliente.
8. El cliente invoca un método de negocio en el stub del EJBObject.
9. El stub del EJBObject se comunica con el EJBObject notificándole la llamada del cliente.
10. Los servicios del contenedor en cuanto a seguridad, transacciones son ejecutados.
11. Se invoca el método de negocio en el bean, para lo cual se saca del pool a un bean. Este bean esta asociado al EJBObject que lo llamo, solo durante el tiempo que dura el la ejecución del método. Una vez ejecutado el método el bean volverá al pool.

3.4.3 Ciclo de vida de un Stateless Session Bean

El ciclo de vida de un stateless session bean consta de dos estados : no existe, y listo.

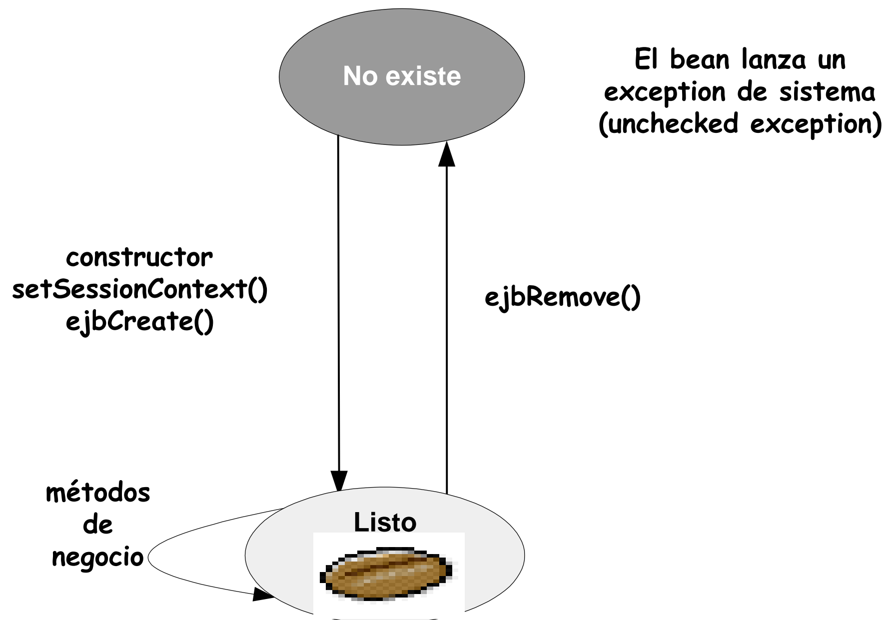


Figura 3.4

No existe — El bean se encuentra en este estado, cuando todavía no ha sido creado por el contenedor. A diferencia de un stateful session bean, una llamada a create(), no crea al stateless session bean.

Listo — El bean se encuentra en este estado una vez que el contenedor invocó al constructor, y lo pone en el pool de beans esperando ser ocupado por un cliente. El contenedor es el que determina cuando debe ser invocado el método ejbRemove() para remover el bean. Los métodos ejbPassivate() y ejbActivate() no aplican para este sabor de sesión bean. Si se intentan invocar los métodos anteriores se lanzará una excepción de tipo EJBException.

3.5 Resumen

Los beans de sesión son ocupados en las aplicaciones empresariales para representar un proceso. Existen dos sabores para este tipo de bean: stateful y stateless. Un stateful session bean es ocupado para mantener un estado conversacional con un cliente, de modo que un bean es asociado sólo con un cliente en un momento dado, y una vez que el cliente ha finalizado su sesión, el bean es destruido para siempre. Un stateless session bean es utilizado para proporcionarle determinada información al cliente, cerrar una venta, hacer un descuento, etc. Este tipo de session bean no puede mantener un estado conversacional con el cliente. El contenedor es el encargado de crear y destruir este tipo de session bean cuando el lo requiera.

Beans de Entidad (Entity Beans)

4.1 Introducción

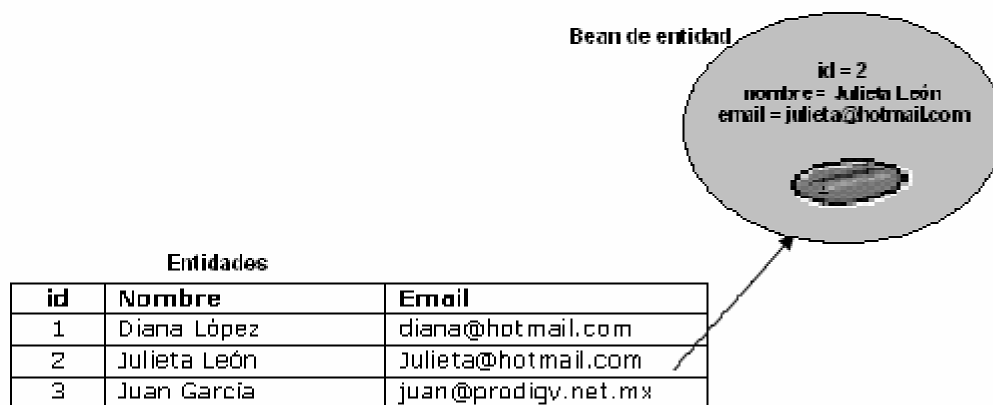
Los beans de entidad son los beans más complejos de los tres tipos de beans que se definen en la arquitectura EJB, tanto por su ciclo de vida, como su construcción. El principal propósito de los beans de entidad es permitir a un cliente operar sobre entidades o registros en la base de datos, por ejemplo, cambiar una dirección email de un consumidor.

Un bean de entidad representa una cosa tal como un empleado, producto u orden, mientras que un bean de sesión representa un proceso tal como la verificación del crédito de una tarjeta de crédito.

4.2 Beans de entidad

Para poder entrar en materia acerca de los beans de entidad, debemos aclarar varios conceptos, los cuales pueden ser confusos, tales como entidad y bean de entidad (entity y entity bean).

Un **entidad** es una cosa que existe en la base de datos, tal como un registro, fuera del contenedor, mientras que un bean de entidad representa a esa entidad en la base de datos. La figura siguiente ilustra el concepto:



id	Nombre	Email
1	Diana López	diana@hotmail.com
2	Julieta León	Julieta@hotmail.com
3	Juan García	juan@prodigy.net.mx

Figura 4.1 Entidades y Beans de entidad

4.2.1 Características de un bean de entidad

En seguida se presentan las algunas características relevantes de los beans de entidad.

- Un bean de entidad se compone de una clase que implementa interface EntityBean y de dos interfaces: home y component.
- Múltiples tipos de beans de entidad pueden ser desplegados en un contenedor.
- Todas las instancias de un particular tipo de bean comparten la misma referencia al objeto que implementa la interface EJBHome, pero cada instancia del bean obtiene su propia referencia al objeto que implementa la interface EJBObject.
- El contenedor provee determinados servicios tales como el manejo de la concurrencia, seguridad y soporte transaccional. El contenedor también se puede encargar del manejo de la persistencia, lo que quiere decir que se encarga del acceso a la base de datos para hacer inserción, actualización, consulta y borrado de registros.

4.2.2 Home Interface

El propósito de los métodos de la interface Home es permitir al cliente hacer ciertas operaciones sobre la entidad, tales como remover una entidad específica, encontrar una entidad específica, etc.

A continuación mostramos los métodos que define la interface EJBHome.

<<interface>>
javax.ejb.EJBHome

```
public EJBMetaData getEJBMetaData()
public HomeHandle getHomeHandle()
public void remove( Handle h )
public void remove( Object key )
```

Aquí esta una breve descripción de los métodos de la interface home, y como debemos utilizarlos en los beans de entidad:

getEJBMetaData() : Este método es utilizado por el cliente para obtener información acerca de la clase del bean.

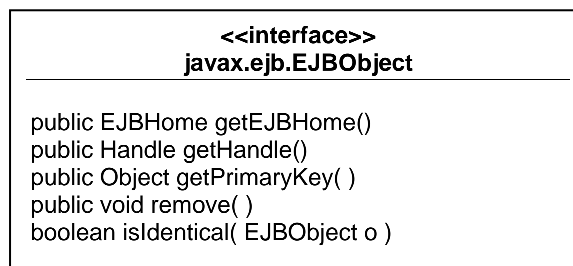
getHomeHandle() : Este método puede ser usado por un cliente para obtener un manejador de la interface home, de modo que si el cliente necesita obtener una referencia al objeto EJBHome de nuevo, no tenga que hacerlo a través de JNDI, si a través de este manejador.

remove(Handle h) : Este método puede ser utilizado por un cliente para remover un entidad.

remove(Object key) : Este método puede ser utilizado por un cliente para remover una entidad, y recibe como argumento la llave primaria de la entidad, la cual será removida de la base de datos.

4.2.3 Component Interface

La interface Component que hereda de la interface EJBObject es utilizada para exponer los métodos de negocio que se escribirán en la clase del bean. El cliente nunca accede directamente al bean, solo lo hace a través de un objeto de implementa la interface EJBObject. La interface EJBObject es utilizada tanto para los beans de sesión como de entidad. La interface EJBObject es presentada a continuación:



La descripción de para que se utilizan estos métodos es la siguiente:

getEJBHome() : Este método regresa una referencia a un objeto de tipo EJBHome. Esto es útil cuando solo se cuenta con la component interface y se quiere obtener una referencia a objeto que implementa la interface EJBHome.

getHandle() : Este método regresa un manejador para el objeto de tipo EJBObject. Este manejador me permite en un momento dado recuperar los objetos EJBObject y EJBHome.

getPrimaryKey() : Este método regresa la llave primaria de la entidad que esta representando el entity bean.

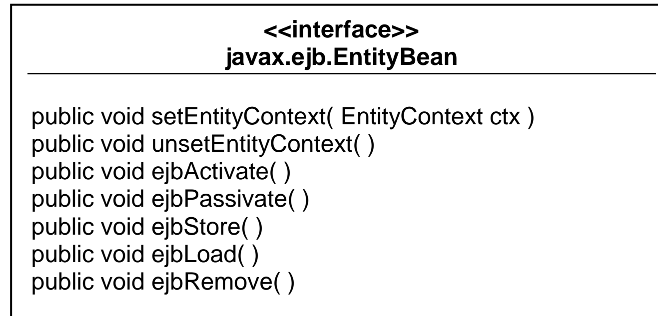
remove() : Este método se encarga de borrar la entidad a la que esta asociado en entity bean.

isIdentical() : Este método sirve para probar si dos objetos de tipo EJBObject son idénticos. Este método regresa **true** para los beans de entidad, si dos stubs se refieren al mismo bean de entidad, el cual representa una entidad en la base de datos, de lo contrario regresa **false**.

4.2.4 EntityBean interface

Para construir un bean de entidad debemos implementar la interface EntityBean. Los métodos que se encuentran en la interface EntityBean son usados por el contenedor para notificar al bean los eventos que suceden a lo largo de su ciclo de vida.

La interface EntityBean es presenta a continuación:



La descripción de los métodos es la siguiente:

setEntityContext() : Este método es usado para pasar una referencia al contexto de entidad a la instancia del bean.

unsetEntityContext() : Invocado por el contenedor para enviar una instancia del bean de entidad del pool al estado de no existencia.

ejbActivate() : Es invocado cuando es bean es sacado del pool para ejecutar un método de negocio que el cliente invoco.

ejbPassivate() : Es invocado antes de enviar a la instancia del bean de entidad de regreso al pool

ejbStore() : Llamado por el contenedor antes de actualizar la entidad en la base de datos con el estado del actual bean.

ejbLoad() : Llamado por el contenedor después de refrescar el estado del bean con los campos de la entidad de la base de datos.

ejbRemove() : Este método es invocado por el contenedor en respuesta a la invocación del cliente del método remove() en el objeto que implementa la interface EJBObject. Este método se encarga de borrar la entidad que representa el bean en la base de datos.

Campos persistentes virtuales

Los campos en un bean de entidad que también deben existir en la base de datos son llamados campos de persistencia. Los valores de estos campos constituyen el estado del bean. Si queremos que los campos sean manejados por el contenedor, ellos no deben ser definidos como variables de instancia en la clase del entity bean. Desde la perspectiva del desarrollador, estos campos son virtuales, ya no existen variables de instancia para ellos, y solo a través de métodos ***get*** y ***set*** pueden ser accesados y modificados. Estos métodos get y set se deben declarar como métodos abstractos.

4.3 Desarrollo de un Entity Bean

De forma parecida al desarrollo de un stateful session bean, el desarrollo de un entity bean también implica la creación de dos interfaces, una que herede la interface EJBHome, y otra que hereda la interface EJBObject, y finalmente, una clase que implementa la interface EntityBean.

A continuación desarrollaremos un entity bean que realizará la inserción de un cliente y sus datos a la base de datos. Como primer paso escribimos la interface Home.

Home Interface : ConsumidorHome.java

```
package com.consumidor;

import javax.ejb.FinderException;
import javax.ejb.CreateException;
import java.rmi.*;

public interface ConsumidorHome extends javax.ejb.EJBHome{
   public ConsumidorComponent create( String rfc, String nombre, String apellidoP, String tel )
          throws CreateException, RemoteException;
   public ConsumidorComponent findByPrimaryKey( String rfc )
          throws FinderException, RemoteException;
}
```

Las reglas para escribir la interface home son las siguientes:

- Se debe importar javax.ejb.* y java.rmi.RemoteException.
- Declarar un método create() con el component interface como valor de retorno. Los métodos create() son utilizados para crear entidades en la base de datos. Por cada método create() que se declare en la interface Home, se debe declarar un método ejbCreate() y un método ejbPostCreate() en la clase que implementa la interface EntityBean.
- El método o los métodos create() deben declarar las excepciones CreateException y RemoteException.
- Se debe declarar un método **findByPrimaryKey()**, el cual recibe como argumento una llave primaria. Este método es implementado por el contenedor.

A continuación mostramos la component interface y sus reglas:

Component Interface : ConsumidorComponent.java

```
package com.consumidor;

import javax.ejb.*;
import java.rmi.*;

public interface ConsumidorComponent extends EJBObject {
    public void setRfc( String rfc ) throws RemoteException;
    public String getRfc( ) throws RemoteException;
    public void setNombre( String nombre ) throws RemoteException;
    public String getNombre() throws RemoteException;
    public void setApellidoP( String apellidoP ) throws RemoteException;
    public String getApellidoP( ) throws RemoteException;
    public void setTelefono( String telefono ) throws RemoteException;
    public String getTelefono( ) throws RemoteException;
}
```

Las reglas para escribir la interface component son las siguientes:

- Se debe importar javax.ejb.*.
- Se debe importar java.rmi.RemoteException.
- Se debe heredar de javax.ejb.EJBObject.
- Los métodos de negocio declarados en la interface deben tener igual signatura o firma que los métodos de negocio que están definidos en la clase del bean.
- Todos los métodos se deben declarar public y no debe haber ningún método declarado static o final.

La clase que define al bean de entidad es la siguiente:

Entity Bean: ConsumidorBean.java

```
package com.consumidor;

import javax.ejb.*;

public abstract class ConsumidorBean implements EntityBean{
    private EntityContext ec;

    // Campos Persistentes virtuales
    public abstract void setRfc( String rfc );
    public abstract String getRfc();
    public abstract void setNombre( String nombre );
    public abstract String getNombre();
    public abstract void setApellidoP( String apellidoP );
    public abstract String getApellidoP();
    public abstract void setTelefono( String telefono );
    public abstract String getTelefono();
```

```
    // Métodos del ciclo de Vida
    public String ejbCreate(String rfc, String nombre, String apellidoP, String tel)
       throws CreateException
    {
       this.setRfc( rfc );
       this.setNombre( nombre );
       this.setApellidoP( apellidoP );
       this.setTelefono( tel );
       return null;
    }

    public void ejbPostCreate(String rfc, String nombre, String apellidoP, String tel){
    }

    public void setEntityContext( EntityContext ec ){
       this.ec = ec;
    }

    public void ejbActivate(){
    }

    public void ejbPassivate(){
    }

    public void ejbRemove(){
    }

    public void unsetEntityContext(){
    }

    public void ejbLoad( ){
    }

    public void ejbStore( ){
    }
}
```

Las reglas para escribir la clase del bean son las siguientes:

- La clase debe implementar la interface javax.ejb.SessionBean.
- Por cada método create() en la home interface debe existir un método ejbCreate() y un método ejbPostCreate() en clase del bean.
- Debe tener un constructor sin argumentos.
- Los métodos de negocio se deben definir en esta clase, pero no tienen que empezar con "ejb", ya que todos los métodos de esta clase que empiezan con ejb controlan el ciclo de vida del bean.
- Por cada campo que tiene la entidad de la base de datos que representa el entity bean, debe tener su campo persistente virtual, el cual se define con los métodos **get<Campo>** y **set<Campo>**, los cuales deben ser **abstract**.

En el descriptor de despliegue donde se indica quien que encargará de escribir el código de acceso a la base de datos, si el desarrollador o el contenedor. También en el descriptor de despliegue se deben mostrar los campos que son mapeados de la base datos al bean e indicamos cual es la llave primaria.

El descriptor de despliegue de este bean se muestra enseguida:

Deployment Descriptor : ejb-jar.xml

```
<?xml version="1.0"?>
<ejb-jar
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                    http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
    version="2.1">
 <enterprise-beans>
  <entity>
     <ejb-name>ConsumidorEJB</ejb-name>
     <home>com.consumidor.ConsumidorHome</home>
     <remote>com.consumidor.ConsumidorComponent</remote>
     <ejb-class>com.consumidor.ConsumidorBean</ejb-class>
     <persistence-type>Container</persistence-type>
     <prim-key-class>java.lang.String</prim-key-class>
     <primkey-field>rfc</primkey-field>
     <reentrant>False</reentrant>

     <abstract-schema-name>Consumidor</abstract-schema-name>
     <cmp-field>
        <field-name>rfc</field-name>
     </cmp-field>
     <cmp-field>
        <field-name>nombre</field-name>
     </cmp-field>
     <cmp-field>
        <field-name>apellidoP</field-name>
     </cmp-field>
     <cmp-field>
        <field-name>telefono</field-name>
     </cmp-field>
  </entity>
 </enterprise-beans>
</ejb-jar>
```

En este descriptor de despliegue se encuentran definidas la siguientes etiquetas:

<home> : se utiliza para indicar cual es la home interface.

<remote> : se utiliza para indicar cual es la component interface

<ejb-class> : se utiliza para indicar cual es la clase del bean.

<persistence-type> : indica quien se encargara del manejo de la persistencia o acceso a base de datos.

<prim-key-class> : indica el tipo de clase a la cual pertenece la llave primaria.

<primkey-field> : indica cual es el campo que es llave primaria.

<cmp-field> : etiqueta dentro de la cual se indicará un campo de la base de datos.

<field-name> : indica cual es el nombre del campo de la base de datos.

Pasaremos a construir un cliente para este bean, el cual es definido como sigue:

Client: ConsumidorClient.java

```
package com.client;

import com.consumidor.*;
import javax.rmi.*;
import javax.naming.*;
import javax.swing.*;

public class ConsumidorClient {
   public static void main(String arg[]){
      try{
         Context ctx = new InitialContext();
         Object obj = ctx.lookup("consumidor");
         ConsumidorHome ch =
                        (ConsumidorHome)PortableRemoteObject.narrow(obj, ConsumidorHome.class);

         String rfc = JOptionPane.showInputDialog( null, "Dame el RFC" );
         String nom = JOptionPane.showInputDialog( null, "Dame el nombre" );
         String ape = JOptionPane.showInputDialog( null, "Dame el apellido" );
         String tel = JOptionPane.showInputDialog( null, "Dame el telefono");
         ConsumidorComponent cc = ch.create(rfc, nom, ape, tel);

         String resumen = "Los datos introducidos fueron:\n";
         resumen += "\nRFC : " + cc.getRfc();
         resumen += "\nNombre : " + cc.getNombre();
         resumen += "\nApellido : " + cc.getApellidoP();
         resumen+= "\nTelefono : " + cc.getTelefono();

         JOptionPane.showMessageDialog( null, resumen );

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

Los pasos que el cliente siguió para poder invocar los métodos de negocio del bean son listados a continuación:

- El cliente hace un búsqueda del objeto home a través de JNDI, el cual es asociado al nombre **"consumidor"**.
- El servicio de JNDI regresa un stub al cliente del Objeto Home.
- El cliente llama al método create() a través del stub. La invocación de este método da como resultado la inserción de los datos en la base de datos.
- El stub envía la llamada al objeto Home, que se encuentra en el servidor.
- El contenedor entonces crea un instancia del bean y del EJBObject.
- El stub para el EJBObject es regresado al cliente.

- El cliente utiliza el EJBObject para mostrar los resultados que fueron insertados en la base de datos.

4.4 Resumen

Los beans de entidad sirven para representar una entidad de una base de datos. Sirven para representar una cosa tal como un libro, un consumidor, un vendedor etc. La principal ventaja del uso de este tipo de bean es que el contenedor se puede encargar del acceso a la base de datos para realizar operaciones tales como insertar, consultar, actualizar y remover. Otra de las funciones del contenedor es asegurarse que solo un cliente del bean de entidad en un momento dado realice una transacción, de modo que es el contenedor el que maneja la concurrencia.

La configuración de a que base de datos se deben insertar los datos, es hecha en el servidor de EJB, de modo que cuando nosotros desarrollamos un bean de entidad no nos preocupamos acerca de la base de datos, fomentando de esta manera la reutilización de componentes.

Beans para el manejo de mensajes (Message Driven beans)

5.1 Introducción

El desarrollo de aplicaciones basada en componentes, implica que necesita haber comunicación. Comunicación envuelve el intercambio de datos entre dos aplicaciones o entre dos componentes de la misma aplicación, y puede ser de dos tipos: *síncrona o asíncrona*. En la comunicación **síncrona**, ambas partes que se están comunicando tienen que estar presentes al mismo tiempo. Por ejemplo, una conversación telefónica, donde las dos personas tienen que estar presentas para entablar una comunicación. La arquitectura EJB ofrece beans de entidad y de sesión para soportar comunicación síncrona, lo cual implica lo siguiente:

- Cuando un cliente invoca un método en una instancia del bean, la instancia del bean debe estar presente en el tiempo que se efectuó la invocación.
- Cuando un cliente realiza una invocación a una instancia de un bean, el debe esperar hasta que la llamada termine sin importar el tiempo para continuar con las siguientes tareas.

En la comunicación **asíncrona** no es necesario que las partes involucradas en la comunicación estén presentes al mismo tiempo. El caso de enviar un correo a una persona en otro país es un caso de comunicación asíncrona. En algunas aplicaciones es necesario este tipo de comunicación, como el envío de correos electrónicos. EJB 2.0 introdujo los beans para el manejo de mensajes (MDB-Message Driven beans) para soportar la comunicación asíncrona en aplicaciones empresariales. Esta comunicación asíncrona es posible gracias al Servicio de Mensajería de Java (JMS-Java Messaging Service).

5.2 Comunicación asíncrona con Beans para el manejo de mensajes

Los beans para el manejo de mensajes utilizan JMS para recibir mensajes. JMS es una API que puede ser usada en la plataforma J2EE para el servicio de mensajería en aplicaciones en empresariales. Una aplicación que envía un mensaje es llamado ***cliente JMS***. Un cliente JMS que genera mensajes es llamado ***productor JMS***, y la aplicación o componente que recibe el mensaje es llamado ***consumidor JMS.*** Entre el productor y el consumidor se encuentra el servidor JMS. El servidor JMS recibe los mensajes de los clientes JMS y los almacena, para después enviarlos al consumidor de ese mensaje.

5.2.1 Funcionamiento de los beans para el manejo de mensajes

La arquitectura EJB ofrece un ambiente asíncrono para que el cliente puede comunicarse con un MDB a través de JMS. La arquitectura y el proceso se muestra en la figura 5.1.

Los pasos que se llevan a cabo durante el proceso son los siguientes:

1. Un cliente, que es el productor, envía un mensaje a la cola (queue) del servidor JMS. El cliente continua después de esto ejecutando sus siguientes líneas de código, sin esperar respuesta del servidor JMS.
2. El servicio de mensajería entrega el mensaje al contenedor EJB.
3. El contenedor notifica al servicio de mensajería (no al productor del mensaje) que el mensaje ha sido recibido. Después, el servicio de mensajería puede remover el mensaje de la cola (queue), pero no del servidor, esto debido a que puede necesitarse de nuevo el mensaje.
4. El contenedor selecciona una instancia del pool de MDB e invoca el método ***onMessage(msg)*** pasándole el mensaje como argumento. El método *onMessage(msg)* es el único método de negocio de los MDB.

Subsecuentemente, el MDB procesa o consume el mensaje, si el proceso falla, el contenedor le dice al servicio de mensajería que vuelva a poner el mensaje de regreso en la cola, y el contenedor colocará el MDB de nuevo en el pool de MDB. Observamos también de este proceso que el cliente nunca invoca directamente un método en el MDB, sino que el mensaje es enviado al servidor JMS, y es el servidor JMS quien pasa el mensaje al contenedor EJB y este último pasa el mensaje al MDB. Debido a lo anterior, se dice que los MDB no tienen clientes, ya que los únicos clientes que hay son del servidor JMS. Como un MDB no tiene un cliente, no es necesario que tenga la interface home y component de los beans de sesión e identidad.

Arquitectura y proceso de la comunicación asíncrona con MDB.

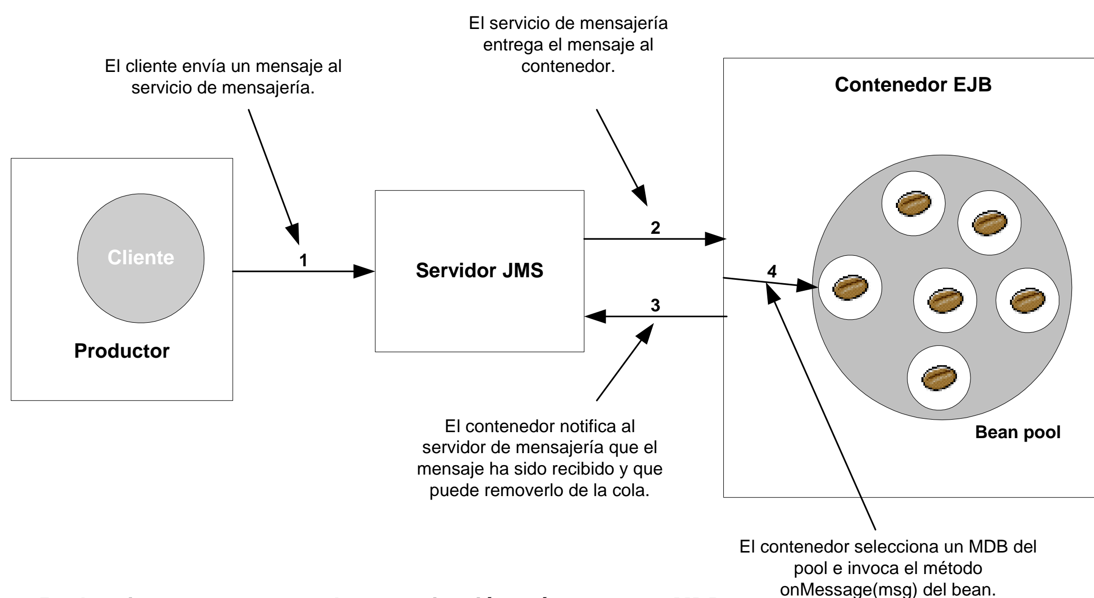
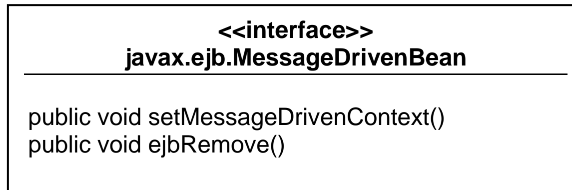


Figura 5.1 Arquitectura y proceso de comunicación asíncrona con MDB

5.2.2 Desarrollo de un Message Driven Bean

Para el desarrollo de un MDB, necesitamos construir una clase, la cual debe implementar dos interfaces: **MessageDrivenBean y MessageListener**. La interface MessageDrivenBean contiene los métodos que el contenedor manda a llamar para controlar su ciclo de vida, mientras la interface MessageListener contiene el único método de negocio que el contenedor invoca cuando recibe un mensaje del servicio de mensajería.



En nuestro ejemplo desarrollaremos un MDB que su único propósito es imprimir el mensaje que recibió.

Message Driven Bean : CustomerNotificationBean.java

```
package com.notificacion;

import javax.ejb.*;
import javax.jms.*;

public class CustomerNotificationBean implements MessageDrivenBean, MessageListener{

    private MessageDrivenContext context = null;

    public void setMessageDrivenContext( MessageDrivenContext ctx ){
        this.context = ctx;
    }
    public void ejbCreate() {
        System.out.println("Ejecutando ejbCreate() de CustomerNotificationBean");
    }
    public void ejbRemove(){
        System.out.println("Ejecutando ejbRemove() de CustomerNotificationBean");
    }
    public void onMessage( Message msg ){
        System.out.println("Ejecutando onMessage()");
        try{
            if( msg instanceof TextMessage)
            {
                TextMessage txtMsg = (TextMessage)msg;
                System.out.println("El mensaje es : " + txtMsg.getText().toUpperCase());
            }
            else
                System.out.println("No es un mensaje de texto!");

        }catch(JMSException ex){
            ex.printStackTrace();
        }
    }
}
```

Descriptor de Despliegue : ejb-jar.xml

```
<?xml version="1.0"?>
<ejb-jar
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd"
    version="2.1">
 <enterprise-beans>
   <message-driven>
      <ejb-name>CustomerNotification</ejb-name>
      <ejb-class>com.notificacion.CustomerNotificationBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
         <destination-type>javax.jms.Queue</destination-type>
      </message-driven-destination>
   </message-driven>
 </enterprise-beans>
</ejb-jar>package com.notificacion;
```

Cuando se desea desarrollar un MDB se deben cumplir con determinados requisitos para asegurar que nuestro MDB va a ser portable para los distintos contenedores EJB.

Requerimientos para la definición de la clase que define un MDB

- La clase debe implementar la interface **javax.ejb.MessageDrivenBean**.
- La clase debe implementar la inteface **javax.jms.MessageListener**.
- La clase debe ser public, no debe ser abstract ni final.

Requerimientos para los métodos

- Se deben implementar los métodos **ejbRemove()** y **setMessageDrivenContext()** de la interface MessageDrivenBean. Los métodos deben ser public, no static ni final.

- Se debe implementar el método **onMessage()** de la inteface **javax.jms.MessageListener.**

- Aunque el método **ejbCreate()** no pertenece a ninguna interface, se debe definir en la clase, ya que este método es invocada por el contenedor para llevar a cabo el ciclo de vida del MDB.

- La clase debe definir un constructor sin argumentos que sea public. Si no se define ningún constructor, el compilador suministrara un constructor sin argumentos. El contenedor usa este constructor para crear una instancia del bean.

- Los métodos en la clase no deben lanzar ninguna **checked exception**, ya que como no hay ningún cliente que maneje estas excepciones.

Se destaca el hecho de que una clase que defina un MDB debe implementar dos interfaces, mientras que los beans de sesión y los beans de entidad solo implementan una interfaz, además de que los métodos del MDB no deben lanzar **checked exceptions**.

Después de definir la clase del MDB, crearemos un cliente para el servicio de mensajería de Java (JMS), el cual recibirá el mensaje y se lo enviará al contenedor EJB para que este se lo pase finalmente al MDB.

Cliente del Servicio de Mensajería (JMS) : NotificationSender.java

```
package com.clients;

import com.notificacion.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.jms.*;

public class NotificationSender {
   public static void main(String arg[]){
      try{
         Context ctx = new InitialContext();
         QueueConnectionFactory factory =
                  (QueueConnectionFactory)ctx.lookup("ConnectionFactory");

         Queue cola = (Queue)ctx.lookup( "queue/NotificarQueue" );
         QueueConnection connect = factory.createQueueConnection();
         QueueSession session = connect.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
         QueueSender sender = session.createSender( cola );

         TextMessage message = session.createTextMessage();
         message.setText("Ejecutar transacción, Fecha  " + new java.util.Date() );
         sender.send( message);
         connect.close();
      }catch(Exception e){
         e.printStackTrace();
      }
   }
}
```

En esta clase, la cual es un cliente de servidor JMS, primero se hace un búsqueda a través de JNDI para obtener una referencia a instancia de tipo QueueConnectionFactory a través de la cual obtenemos una conexión al servicio de mensajería, para después obtener un sesión y poder enviar los mensajes.

5.3 Ciclo de vida de un MDB

El ciclo de vida de un MDB es muy simple. Solo existen dos estados: el estado de no existencia y el estado pooled o en espera de procesar un mensaje. El ciclo de vida y los estados del MDB se muestra a continuación:

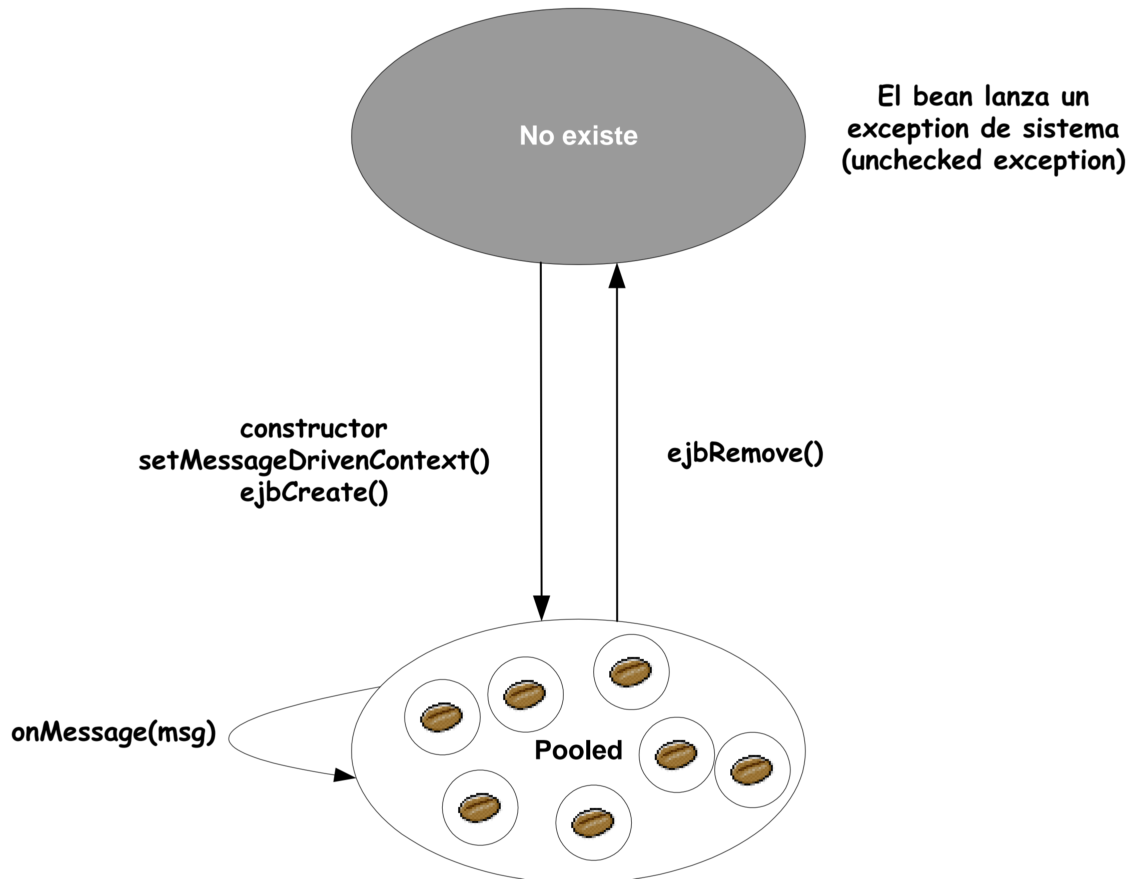


Figura 5.2 Ciclo de vida de un Message Driven Bean

Una instancia de un MDB pasa del estado de no existe a pooled, cuando el contenedor instancia un objeto, por invocar al constructor del MDB. Después de que la instancia es creada, el contenedor invoca a los métodos **setMessageDrivenContext()** y **ejbCreate()**. El contenedor puede crear un número determinado de MDB y ponerlos en el pool. Todos los beans del mismo tipo son colocados en un pool diferente de los demás tipos de beans

Cuando el contenedor quiere remover una instancia, por ejemplo, para ahorrar recursos, invoca al método **ejbRemove()** del bean, entonces este bean pasará al estado de no existencia.

Un MDB puede pasar también al estado de no existencia cuando lanza un excepción de sistema ó **unchecked exception**.

5.4 Tipos de destinación

Un cliente es el que realiza el envío de un mensaje a una destinación JMS, que es una dirección en el servidor JMS. Un message driven bean es asociado con una destinación, y la destinación puede ser de dos tipos: **Topic ó Queue**. Usando el elemento **<message-driven-destination>** en el descriptor de despliegue, el desarrollador del bean indica que tipo de destinación el bean utiliza. A continuación se muestra un fragmento del descriptor de despliegue:

```
<enterprise-beans>
   <message-driven>
      <ejb-name>CustomerNotification</ejb-name>
      <ejb-class>com.notificacion.CustomerNotificationBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
         <destination-type>javax.jms.Queue</destination-type>
      </message-driven-destination>
   </message-driven>
</enterprise-beans>
```

Si el MDB fuera a esperar mensajes de una destinación de tipo Topic, se usaría **javax.jms.Topic** en lugar de **javax.jms.Queue**. En las siguientes secciones se describen los tipos de destinación y sus diferencias.

5.4.1 Destinación de tipo Queue

La interface **javax.jms.Queue** soporta el modelo de mensajes punto-a-punto (point-to-point) en el cual un solo consumidor (MDB) puede recibir un mensaje enviado por un productor (client). Una vez que le mensaje ha sido procesado por una instancia del bean, el mensaje es removido de la cola o queue, y ningún otra instancia del bean puede procesarlo. Una destinación de tipo Queue se asegura de que el consumidor no pierda el mensaje aun si el servidor EJB esta abajo cuando fue recibido el mensaje.

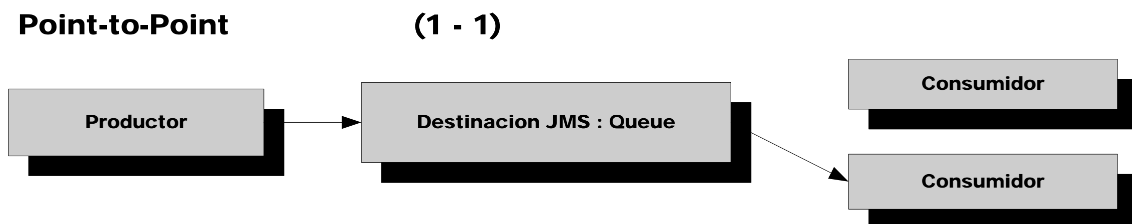


Figura 5.3 Modelo de mensajes point-to-point

5.4.2 Destinación de tipo Topic

La interface **javax.jms.Topic** soporta el modelo de mensajes publish/subscribe en el cual múltiples beans pueden recibir el mismo mensaje. Todos los consumidores que están inscritos en el **Topic** pueden recibir el mensaje enviado por un productor (cliente). Múltiples consumidores (llamados suscriptores) que estén interesados en un tópico pueden inscribirse en el. Cuando un mensaje llega una destinación de tipo Topic, todos los suscriptores de este Topic pueden recibir el mensaje.

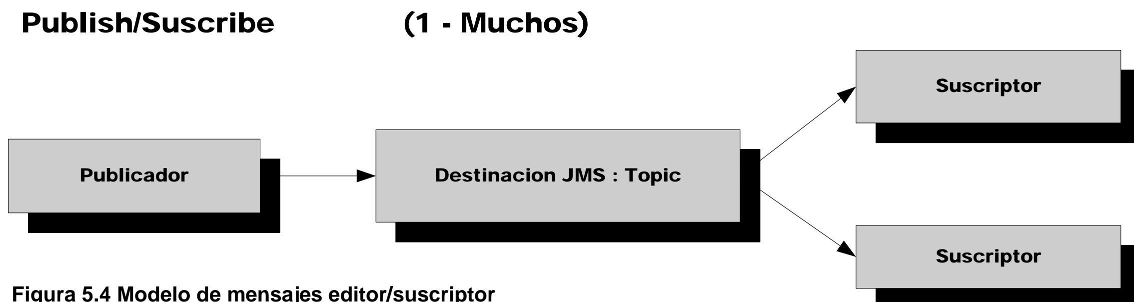


Figura 5.4 Modelo de mensajes editor/suscriptor

5.5 Resumen

La arquitectura EJB ofrece soporte tanto para establecer una comunicación síncrona y asíncrona. Para la comunicación síncrona contamos con los beans de sesión y de entidad. Los MDB, que fueron introducidos en la versión EJB 2.0 soportan la comunicación asíncrona. Los MDBs utilizan el servicio de mensajería de Java (JMS) para llevar a cabo su propósito.

También vimos que para poder construir un MDB, necesitamos implementar dos intefaces: javax.ejb.MessageDrivenBean y javax.jms.MessageListener. Estas interfaces contienen métodos que controlan el ciclo de vida de bean, desde su creación hasta su muerte.

Por último, mostramos que MDB soporta los dos modelos de mensaje: publish/subscribe y point-to-point. Deacuerdo a cuantos beans queremos que reciban un mensaje es el tipo de destinación que elegimos. El tipo de destinación Queue soporta el modelo de mensajes point-to-point, mientras que Topic soporta el modelo de mensajes publish/subscribe.

Despliegue de Enterprise Beans usando el servidor de aplicaciones JBoss

6.1 Introducción

JBoss es un servidor de aplicaciones basado en la edición empresario de la plataforma Java (J2EE). JBoss es actualmente el líder en servidores de aplicaciones, con más de cinco millones de descargas en los últimos dos años.

JBoss implementa todos los servicios y tecnologías que se necesitan en una aplicación J2EE:

- Servles y JSP (JavaServer Pages)
- EJB (Enterprise JavaBeans)
- JMS (Java Message Service)
- JTS/JTA (Java Transaction Service/Java Transaction API)
- JNDI (Java Naming and Directory Interface)
- Web Services

Jboss esta bajo la licencia LGPL (GNU Lesser General Public License), que nos brinda los siguientes beneficios: libertad de uso, sin costo en cualquier aplicación comercial, además es de libre distribución.

6.2 Instalación de JBoss y Ant

Antes de empezar la instalación del servidor de aplicaciones JBoss y de la herramienta Ant, la cual me ayuda en la compilación y despliegue de las aplicaciones Java, se debe tener instalado el J2SE JDK 1.4 o una versión superior.

El servidor de aplicaciones JBoss se puede descargar del siguiente sitio web:

http://www.jboss.org

La herramienta para la construcción de aplicaciones Java se encuentra en el siguiente sitio:

http://jakarta.apache.org/ant/

Una vez que hemos descargado el servidor de aplicaciones y la herramienta ant, procedemos a la instalación, para lo cual se debe descomprimir los archivos.

Unix:

gunzip jboss-4.0.tar.gz
tar xf jboss-4.0.tar

tar zxvf ant-6.0.tar.gz

Windows

En windows se utiliza winzip.

Una vez que hemos descomprimido los archivos, debemos asegurarnos de configurar las siguientes variables de entorno:

JAVA_HOME:C://Ruta del J2SE JDK
JBOSS_HOME=C://Ruta de JBOSS

En la variable de ambiente PATH se debe añadir la siguiente ruta de la herramienta ANT:

Unix:

PATH=$PATH:/Ruta de Ant/bin

Window:

PATH=%PATH%:C://Ruta de Ant/bin

Una vez que hemos hecho lo anterior, procedemos a levantar el servidor de aplicaciones:

Unix:

JBOSS_HOME/bin/run.sh

Windows:

C:\jboss-4.0\bin\run.bat

Después de ejecutar el comando, el servidor JBoss se encuentra trabajando. Para parar el servidor de aplicaciones se utilizan los siguientes comandos:

Unix:

JBOSS_HOME/bin/shutdown.sh

Windows:

C:\jboss-4.0\bin\shutdown.bat

6.2.1 Estrutura de directorios de JBoss

Cuando se instala JBoss, se crea la siguiente estructura de directorios:

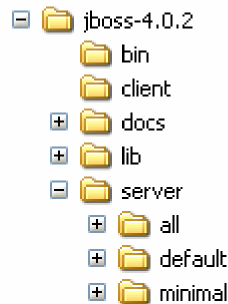


Figura 6.1 Estructura de directorios de JBoss

La descripción de algunos directorios se presenta a continuación:

bin	Scripts para arrancar y parar JBoss.
docs	Ejemplos de archivos de configuración
lib	Archivos JAR que son cargados cuando se arranca JBoss.
server	Se encuentran las distintas configuraciones que puede tener JBoss. La configuración con la que arranca el servidor normalmente es ***default***.

En el directorio **server/default/deploy** es donde se colocan los archivos JAR, WARs y EARs para su despliegue.

En la siguiente sección se procederá a construir y desplegar todos los ejemplos de enterprise beans que fueron desarrollados en cada capítulo, para lo cual utilizaremos el servidor de aplicaciones JBoss, el cual se deberá estar ejecutándose.

6.3 Despliegue del bean de sesión "Quote"

Para poder desplegar nuestro bean de sesión es necesario que construir la siguiente estructura de directorios:

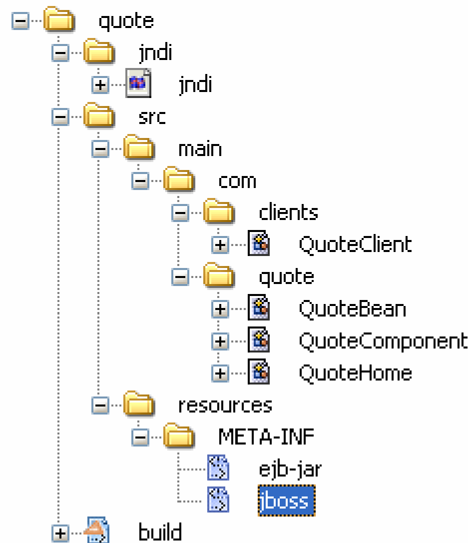


Figura 6.2 Estructura de archivos del bean "Quote"

Las clases e interfaces QuoteClient, QuoteBean, QuoteComponent y QuoteHome fueron definidas en el capitulo 2 junto con el archivo ejb-jar.xml. Los archivos específicos de JBoss para poder desplegar la aplicación son presentados a continuación junto con el archivo de propiedades de JNDI y el archivo de construcción de Ant que se requiere:

El archivo de configuración JNDI es el mismo para todos los ejemplos desarrollados a lo largo de los capítulos, y se presenta a continuación:

jndi.properties

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost
```

Archivo de configuración para el despliegue del bean quote requerido por JBoss.

jboss.xml

```
<?xml version="1.0"?>
<jboss>
    <enterprise-beans>
      <session>
            <ejb-name>QuoteBeanSession</ejb-name>
            <jndi-name>quoter</jndi-name>
      </session>
    </enterprise-beans>
</jboss>
```

Archivo de construcción y despliegue de Ant:

build.xml

```
<?xml version="1.0"?>

<project name="JBoss" default="ejbjar" basedir=".">

  <property environment="env"/>
  <property name="src.dir" value="${basedir}/src/main"/>
  <property name="src.resources" value="${basedir}/src/resources"/>
  <property name="jboss.home" value="${env.JBOSS_HOME}"/>
  <property name="build.dir" value="${basedir}/build"/>
  <property name="build.classes.dir" value="${build.dir}/classes"/>

  <!-- =================================================================== -->
  <!-- Establecer el classpath  y construcción del directorio build        -->
  <!-- =================================================================== -->

  <path id="classpath">
      <fileset dir="${jboss.home}/client">
         <include name="**/*.jar"/>
      </fileset>
            <pathelement location="${build.classes.dir}"/>
            <pathelement location="${basedir}/jndi"/>
  </path>

  <property name="build.classpath" refid="classpath"/>

  <target name="prepare" >
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.classes.dir}"/>
  </target>

  <!-- =================================================================== -->
  <!-- Compilar código fuente                                               -->
  <!-- =================================================================== -->

  <target name="compile" depends="prepare">
    <javac srcdir="${src.dir}" destdir="${build.classes.dir}" debug="on" optimize="off" includes="**">
         <classpath refid="classpath"/>
    </javac>
  </target>

  <target name="ejbjar" depends="compile">
    <jar jarfile="build/quote.jar">
      <fileset dir="${build.classes.dir}">
             <include name="com/quote/*.class"/>
      </fileset>
      <fileset dir="${src.resources}/">
         <include name="**/*.xml"/>
      </fileset>
     </jar>
     <copy file="build/quote.jar" todir="${jboss.home}/server/default/deploy"/>
  </target>

  <!-- =================================================================== -->
  <!-- Invocar la aplicación cliente                                        -->
  <!-- =================================================================== -->

   <target name="cliente" depends="ejbjar">
    <java classname="com.clients.QuoteClient" fork="yes" dir=".">
      <classpath refid="classpath"/>
    </java>
  </target>
</project>
```

Para poder compilar nuestra aplicación y copiar nuestro archivo jar generado, necesitamos colocarnos dentro del directorio **quote**, e invocar el siguiente comando, que buscará nuestro archivo de construcción build.xml, el cual contiene las instrucción de construcción y compilación:

Unix:

ant

Windows:

C:\quote> ant

Una vez ejecutado este comando, nuestro enterprise bean será desplegado en JBoss. Para poder invocar nuestro enterprise bean, necesitamos ejecutar nuestro cliente, lo cual lo podemos hacer con nuestro script build.xml de la siguiente manera:

Unix:

ant cliente

Windows:

C:\quote> ant cliente

Si todo se ejecuta de manera correcta, podremos ver los resultados de invocar al enterprise bean.

6.4 Despliegue del bean de sesión "Beer"

La siguiente estructura de directorios en necesaria para construir nuestra aplicación:

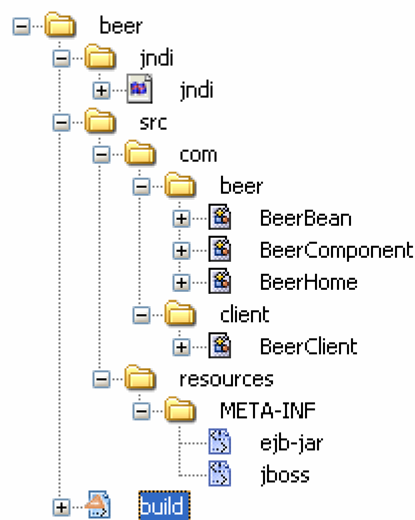


Figura 6.3 Estructura de archivos del bean "Beer"

Los archivos de clase e interfaces que desarrollamos en el capitulo 3, en la sección de beans de sesión de tipo stateful son: BeerBean, BeerComponent, BeerHome, BeerClient y el descriptor de despliegue ejb-jar.xml. A continuación presentamos el archivo de configuración que ocupa JBoss para desplegar nuestro bean, y el script de Ant para construir nuestra aplicación:

Archivo de configuración para el despliegue del bean "beer" requerido por JBoss

jboss.xml

```
<?xml version="1.0"?>

<jboss>
    <enterprise-beans>
      <session>
            <ejb-name>BeerBean</ejb-name>
            <jndi-name>beer</jndi-name>
      </session>
    </enterprise-beans>
</jboss>
```

Script de Ant para la construcción y compilación de nuestra aplicación:

build.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- =================================================================== -->
<!-- Archivo de construcción                                             -->
<!-- =================================================================== -->

<project basedir="." default="ejbjar" name="JBoss">

  <property environment="env"/>
  <property name="src.dir" value="${basedir}/src"/>
  <property name="src.resources" value="${basedir}/src/resources"/>
  <property name="jboss.home" value="${env.JBOSS_HOME}"/>
  <property name="build.dir" value="${basedir}/build"/>
  <property name="build.classes.dir" value="${build.dir}/classes"/>

  <path id="classpath">
      <fileset dir="${jboss.home}/client">
         <include name="**/*.jar"/>
      </fileset>
            <pathelement location="${build.classes.dir}"/>
            <pathelement location="${basedir}/jndi"/>
  </path>

  <property name="build.classpath" refid="classpath"/>

  <target name="prepare">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.classes.dir}"/>
  </target>
```

```
  <!-- =================================================================== -->
  <!-- Compilar código fuente y crear archivo jar                          -->
  <!-- =================================================================== -->
  <target depends="prepare" name="compile">
    <javac debug="on" deprecation="on" destdir="${build.classes.dir}" includes="**" optimize="off" srcdir="${src.dir}">
        <classpath refid="classpath"/>
    </javac>
  </target>

  <target depends="compile" name="ejbjar">
    <jar jarfile="build/beer.jar">
      <fileset dir="${build.classes.dir}">
            <include name="com/beer/*.class"/>
      </fileset>
      <fileset dir="${src.resources}/">
         <include name="**/*.xml"/>
      </fileset>
     </jar>
     <copy file="build/beer.jar" todir="${jboss.home}/server/default/deploy"/>
  </target>

  <target depends="ejbjar" name="cliente">
    <java classname="com.client.BeerClient" dir="." fork="yes">
      <classpath refid="classpath"/>
    </java>
  </target>

  <target name="clean">
    <delete dir="${build.dir}"/>
    <delete file="${jboss.home}/server/default/deploy/beer.jar"/>
  </target>
</project>
```

Una vez que tenemos estos archivos, nos colocamos dentro del directorio beer, y ejecutamos el siguiente comando para construir nuestra aplicación:

Unix:

ant

Windows:

ant

Una vez que hemos construido la aplicación, ejecutamos el cliente:

Unix:

ant cliente

Windows:

C:\beer> **ant cliente**

6.5 Despliegue del bean de sesión "Discount"

La estructura de directorios necesaria para nuestra aplicación se presenta a continuación:

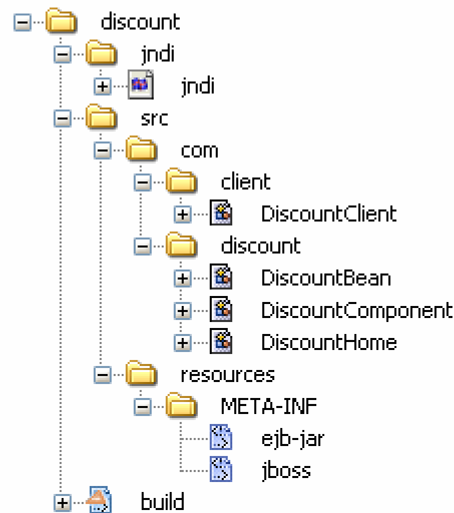


Figura 6.4 Estructura de archivos del bean "Discount"

A continuación se proporciona el archivo de construcción de la aplicación build.xml, y el archivo que requiere jboss para el despliegue del bean.

jboss.xml

```
<?xml version="1.0"?>

<jboss>
    <enterprise-beans>
    <session>
            <ejb-name>Discount Bean</ejb-name>
            <jndi-name>discount</jndi-name>
      </session>
    </enterprise-beans>
</jboss>
```

Archivo de construcción build.xml:

build.xml

```
<?xml version="1.0"?>

<!-- ======================================================================= -->
<!-- Archivo de Construcción                                                  -->
<!-- ======================================================================= -->
```

```
<project name="JBoss" default="ejbjar" basedir=".">

  <property environment="env"/>
  <property name="src.dir" value="${basedir}/src"/>
  <property name="src.resources" value="${basedir}/src/resources"/>
  <property name="jboss.home" value="${env.JBOSS_HOME}"/>
  <property name="build.dir" value="${basedir}/build"/>
  <property name="build.classes.dir" value="${build.dir}/classes"/>

  <path id="classpath">
       <fileset dir="${jboss.home}/client">
          <include name="**/*.jar"/>
       </fileset>
            <pathelement location="${build.classes.dir}"/>
            <pathelement location="${basedir}/jndi"/>
  </path>

  <property name="build.classpath" refid="classpath"/>

  <target name="prepare" >
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.classes.dir}"/>
  </target>

  <!-- =================================================================== -->
  <!-- Compilar código fuente                                              -->
  <!-- =================================================================== -->

  <target name="compile" depends="prepare">
    <javac srcdir="${src.dir}"
           destdir="${build.classes.dir}"
           debug="on"
           deprecation="on"
           optimize="off"
           includes="**">
            <classpath refid="classpath"/>
    </javac>
  </target>

  <target name="ejbjar" depends="compile">
    <jar jarfile="build/discount.jar">
      <fileset dir="${build.classes.dir}">
                <include name="com/discount/*.class"/>
      </fileset>
      <fileset dir="${src.resources}/">
         <include name="**/*.xml"/>
      </fileset>
     </jar>
     <copy file="build/discount.jar" todir="${jboss.home}/server/default/deploy"/>
  </target>

  <target name="cliente" depends="ejbjar">
    <java classname="com.client.DiscountClient" fork="yes" dir=".">
      <classpath refid="classpath"/>
    </java>
  </target>

</project>
```

Una vez que tenemos estos archivos, nos colocamos dentro del directorio discount y ejecutamos el siguiente comando para construir nuestra aplicación:

Unix :

ant

Windows:

ant

Una vez que hemos construido la aplicación, ejecutamos el cliente:

Unix:

ant cliente

Windows:

C:\discount > **ant cliente**

6.6 Despliegue del bean de entidad "Consumidor"

La estructura de directorios para poder ejecutar el bean consumidor es la siguiente:

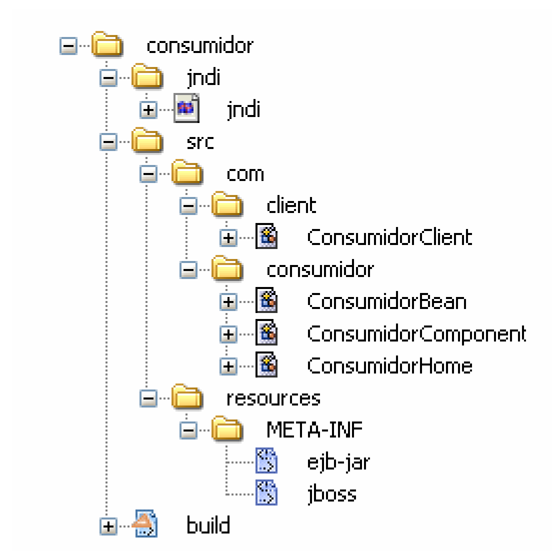


Figura 6.5 Estructura de archivos del bean "Consumidor"

Los archivos jboss.xml y build.xml son presentados a continuación:

jboss.xml

```
<?xml version="1.0"?>

<jboss>
    <enterprise-beans>
      <entity>
            <ejb-name>ConsumidorEJB</ejb-name>
            <jndi-name>consumidor</jndi-name>
      </entity>
    </enterprise-beans>
</jboss>
```

Archivo de construcción:

build.xml

```
<?xml version="1.0"?>

<!-- ====================================================================== -->
<!-- Archivo de Construcción                                                -->
<!-- ====================================================================== -->

<project name="JBoss" default="ejbjar" basedir=".">

  <property environment="env"/>
  <property name="src.dir" value="${basedir}/src"/>
  <property name="src.resources" value="${basedir}/src/resources"/>
  <property name="jboss.home" value="${env.JBOSS_HOME}"/>
  <property name="build.dir" value="${basedir}/build"/>
  <property name="build.classes.dir" value="${build.dir}/classes"/>

  <path id="classpath">
      <fileset dir="${jboss.home}/client">
          <include name="**/*.jar"/>
      </fileset>
            <pathelement location="${build.classes.dir}"/>
            <pathelement location="${basedir}/jndi"/>
  </path>

  <property name="build.classpath" refid="classpath"/>

   <target name="prepare" >
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.classes.dir}"/>
  </target>

  <!-- =================================================================== -->
  <!-- Compilar código fuente                                              -->
  <!-- =================================================================== -->
   <target name="compile" depends="prepare">
    <javac srcdir="${src.dir}"
          destdir="${build.classes.dir}"
          debug="on"
          deprecation="on"
          optimize="off"
          includes="**">
           <classpath refid="classpath"/>
```

```
    </javac>
  </target>

  <target name="ejbjar" depends="compile">
    <jar jarfile="build/consumidor.jar">
      <fileset dir="${build.classes.dir}">
              <include name="com/consumidor/*.class"/>
      </fileset>
      <fileset dir="${src.resources}/">
          <include name="**/*.xml"/>
      </fileset>
     </jar>

     <copy file="build/consumidor.jar" todir="${jboss.home}/server/default/deploy"/>

  </target>

  <!-- Ejecutar el cliente -->
  <target name="cliente" depends="ejbjar">
    <java classname="com.client.ConsumidorClient" fork="yes" dir=".">
      <classpath refid="classpath"/>
    </java>
  </target>

  <!-- Tareas de limplieza -->
   <target name="clean.db">
    <delete dir="${jboss.home}/server/default/data/hypersonic"/>
  </target>

  <target name="clean">
    <delete dir="${build.dir}"/>
    <delete file="${jboss.home}/server/default/deploy/consumidor.jar"/>
  </target>
</project>
```

Una vez que tenemos estos archivos, nos colocamos dentro del directorio consumidor y ejecutamos el siguiente comando para construir nuestra aplicación:

Unix :

ant

Windows:

ant

Al ejecutar este comando, se construye nuestra aplicación, y se copia el archivo consumidor.jar que se creo en el directorio de despliegue de aplicaciones de JBoss. Al ver JBoss es archivo anterior, lee el archivo jboss.xml, y se da cuenta que es un bean de entidad, y como no tiene configurada ninguna base de datos para el despliegue de este bean, crea las tablas necesarias en la base embebida Hipersonic. Los campos que construye, están basados en el descriptor de despliegue ejb-jar.xml del bean. Una vez que hemos construido la aplicación, ejecutamos el cliente:

Unix:

ant cliente

Windows:

C:\consumidor > **ant cliente**

6.7 Despliegue del bean para el manejo de mensajes "Notificar"

La estructura de directorios requerida para poder desplegar nuestro bean es la siguiente:

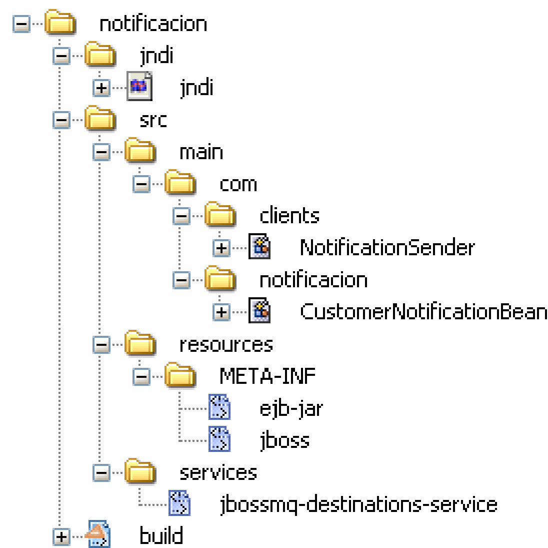


Figura 6.6 **Estructura de archivos del bean "Notificar"**

Para poder desplegar el bean para el manejo de mensajes se requiere otro archivo donde se configura que tipo de destinación, ya sea Queue o Topic.

Este archivo es presentado a continuación:

jbossmq_destinations-service.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<server>
  <mbean code="org.jboss.mq.server.jmx.Queue"
              name="jboss.mq.destination:service=Queue,name=NotificarQueue">
        <depends optional-attribute-name="DestinationManager">
                              jboss.mq:service=DestinationManager
       </depends>
  </mbean>
</server>
```

El archivo jboss.xml es presentado a continuación:

jboss.xml

```xml
<?xml version="1.0"?>

<jboss>
    <enterprise-beans>
      <message-driven>
          <ejb-name>CustomerNotification</ejb-name>
          <destination-jndi-name>queue/NotificarQueue</destination-jndi-name>
      </message-driven>
    </enterprise-beans>
</jboss>
```

En seguida se presenta el archivo de construcción build.xml:

bulid.xml

```xml
<?xml version="1.0"?>

<!-- ====================================================================== -->
<!-- Archivo de Construcción                                                -->
<!-- ====================================================================== -->

<project name="JBoss" default="ejbjar" basedir=".">

  <property environment="env"/>
  <property name="src.dir" value="${basedir}/src/main"/>
  <property name="src.resources" value="${basedir}/src/resources"/>
  <property name="jboss.home" value="${env.JBOSS_HOME}"/>
  <property name="build.dir" value="${basedir}/build"/>
  <property name="build.classes.dir" value="${build.dir}/classes"/>

  <path id="classpath">
      <fileset dir="${jboss.home}/client">
          <include name="**/*.jar"/>
      </fileset>
              <pathelement location="${build.classes.dir}"/>
              <pathelement location="${basedir}/jndi"/>
  </path>

  <property name="build.classpath" refid="classpath"/>

  <target name="prepare" >
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.classes.dir}"/>
  </target>

  <!-- ================================================================== -->
  <!-- Compilar código fuente                                             -->
  <!-- ================================================================== -->

  <target name="compile" depends="prepare">
    <javac srcdir="${src.dir}"
           destdir="${build.classes.dir}"
           debug="on"
           deprecation="on"
           optimize="off"
           includes="**">
            <classpath refid="classpath"/>
    </javac>
  </target>
```

```
 <target name="ejbjar" depends="compile">
   <jar jarfile="build/notification.jar">
     <fileset dir="${build.classes.dir}">
             <include name="com/notificacion/*.class"/>
     </fileset>
     <fileset dir="${src.resources}/">
         <include name="**/*.xml"/>
     </fileset>
    </jar>
    <copy file="build/notification.jar" todir="${jboss.home}/server/default/deploy"/>
 </target>

 <target name="colas">
   <copy file="src/services/jbossmq-destinations-service.xml" todir="${jboss.home}/server/default/deploy"/>
  </target>
 <!-- Ejecutar el cliente -->
 <target name="cliente" depends="ejbjar">
   <java classname="com.clients.NotificationSender" fork="yes" dir=".">
     <classpath refid="classpath"/>
   </java>
 </target>

 <!-- Tareas de limplieza -->
 <target name="clean">
   <delete dir="${build.dir}"/>
   <delete file="${jboss.home}/server/default/deploy/notification.jar"/>
 </target>
</project>
```

Los pasos que se presentan a continuación, son los mismos que se utilizan para construir y ejecutar las aplicaciones anteriores:

Unix :

ant

Windows:

ant

Al ejecutar este comando, se construye nuestra aplicación, y se copia el archivo notificacion.jar que se creo en el directorio de despliegue de aplicaciones de JBoss. Para ejecutar el cliente se escribe el siguiente comando

Unix:

ant cliente

Windows:

C:\consumidor > **ant cliente**

Conclusiones

El desarrollo de aplicaciones las cuales tengan como requerimiento el poder ser extensibles, distribuidas, y una disponibilidad alta ha sido creciente en los últimos años. Lo anterior provoca que se desarrollen soluciones para este problema, entre las que se encuentra la arquitectura J2EE, la cual contiene la arquitectura EJB. La arquitectura EJB nos proporciona los componentes que nos ayudan a desarrollar aplicaciones distribuidas. Los componentes EJB son : Session Beans, Entity Beans, y Message-Driven Beans. Cada uno de estos componentes juegan un papel específico en la aplicación empresarial, los Session Beans son utilizados para representar un proceso, tal como efectuar un descuento, validar información acerca de tarjetas de crédito, etc. mientras los Entity Beans representan entidades o registros de una base de datos, y se encargan de toda la parte transaccional, en último lugar tenemos a los Message Driven Beans, los cuales son utilizados para establecer una comunicación asíncrona.

La arquitectura EJB, esta basada en componentes, los cuales nos facilitan desarrollar aplicaciones distribuidas, estos componentes no obstante, no son fáciles de desarrollar, ya que presentan cierto grado de complejidad, principalmente los Entity Beans, ya que tienen ciertas restricciones en su desarrollo, las cuales son esenciales para su correcto funcionamiento, debido a esta complejidad en el esquema de persistencia de los Entity Bean, han surgido nuevas tecnologías, talas como Hibernate y Castor.

La capa de negocios, dentro de la cual entran los enterprise beans, se puede integrar fácilmente con la capa web, de modo que podamos tener cada capa en máquinas distintas.

Hemos aprendido que para el despliegue de Enterprise beans, es necesario utilizar un servidor de aplicaciones, el cual proporciona el contenedor EJB. Entre los servidores de aplicaciones más importantes, encontramos a JBoss, el cual nos proporciona un entorno fácil de administrar, y en el cual podemos ver el despliegue de los Enterprise Beans.

Referencias

- Kathy Sierra,Bert Bates, **Head First EJB**, ***Passing the Sun Certified Business Component Developer Exam***, editorial O'REILLY, Primera Edición, 2003.

- Paul Sanghera, **SCBCD**, ***Java Business Component Developer Certification For EJB***, editorial Manning, Primera Edición, 2005.

- Richard Monson-Haefel, **Enterprise JavaBeans**, ***Developing Enterprise Java Components***, editorial O'REILLY, Cuarta Edición, 2004.

- Jim Keogh, **J2EE, Manual de referencia**, editorial Mc Graw Hill, primera edición, 2004.

- Kyle Brown, Gary Craig, Greg Hester, Russell Stinehour, W. David Pitt, Mark Weitzel, Jim Amsden, Peter M. Jakab, Daniel Berg, **Enterprise Java Programming with IBM Websphere**, editorial Addison Wesley, Segunda Edición, 2003.

- Cay S. Horstmann, Gary Cornell, **Core JAVA 2**, ***Volume II-Advanced Features***, editorial Prentice Hall, Septima Edición, 2005.

- Norman Richards, Sam Griffith, **JBoss**, ***A Developer's Notebook***, editorial O'REILLY, primera edición, 2005.

- Ken Arnold, James Gosling, David Holmes, **The Java Programming Language**, editorial Addison Wesley, Cuarta Edición, 2005.

- Harvey M. Deitel, Paul J. Deitel, **Cómo Programar en JAVA**, editorial Prentice Hall, Quinta edición, 2004.

- Christian Bauer, Gavin King, **HIBERNATE** ***in Action***, editorial Manning, Primera edición, 2005.

- Deepak Alur, John Crupi, Dan Malks, **Core J2EE Patterns**, ***Best Practices and Design Strategies***, editorial Prentice Hall, Segunda edición, 2003.